Chapter 1

STRING PATTERN MATCHING FOR A DELUGE SURVIVAL KIT

Alberto Apostolico
Department of Computer Sciences, Purdue University, USA and *Dipartimento di Elettronica e Informatica, Università di Padova, Italy.*
and Maxime Crochemore
Institut Gaspard Monge, Université de Marne-la-Vallée, Noisy-le-Grand, France.

Abstract String Pattern Matching concerns itself with algorithmic and combinatorial issues related to matching and searching on linearly arranged sequences of symbols, arguably the simplest possible discrete structures. As unprecedented volumes of sequence data are amassed, disseminated and shared at an increasing pace, effective access to, and manipulation of such data depend crucially on the efficiency with which strings are structured, compressed, transmitted, stored, searched and retrieved. This paper samples from this perspective, and with the authors' own bias, a rich arsenal of ideas and techniques developed in more than three decades of history.

1. INTRODUCTION

This chapter reviews a number of rather ubiquitous primitives related to matching and searching with some elementary discrete structures such as strings, regular expressions, and other aggregates, that are likely to be of relevance, directly or indirectly, in the current and future infrastructures of very large volumes of data. In that context, massive, scattered and diverse information repositories will pose increasing needs for novel approaches to their management by means of compression, inference, comparison and retrieval, mining, and related principles and techniques. Without pretending to be exhaustive, the selection of topics presented in this chapter was inspired by two main principles beside the authors' own bias. The first one, was to recognize that the data flood is forcing a paradigm shift to take place, whereby the previous ambition to organize and funnel to the user as much data as possible is being changed into that of limiting and filtering what the limited ultimate bandwidth, the user himself, may actually intake. The second, and related principle, is that, in computer science jargon, search by value is going to be increasingly replaced by search by contents and, in turn, by search by meaning, in the future. It is believed that, while eminently syntactic in nature, most of the primitives considered here shall still form the core of the semantic capabilities subtending automated association generation and other similar techniques of filtration and inference.

Problems of matching and searching, and the combinatorial properties that support their efficient solutions, may be classified according to a number of paradigms. One way to classify these problems is according to the type of structure (strings, arrays, trees, etc.) in terms of which they are posed. Another is according to the model of computation used, *e.g.*, serial or parallel. Yet another one is according to whether the manipulations that one seeks to optimize need be performed on-line, off-line, in real time, etc. One could distinguish further between matching and searching and, within the latter, between exact and approximate searches, or vice versa. The classification used here privileges certain aspects of exact or approximate searching, combinatorial issues such as the identification of periodicities, symmetries and other regularities, efficient implementations of ancillary functions such as compression and encoding, etc., that are perceived as most relevant in the current context. Due to space limitations we emphasize here problems on strings, but it should be clear that most problems (albeit not their solutions) translate straightforwardly to more complicated structures.

This chapter is organized as follows. In the next section, we review some fundamental facts about regularities that manifest themselves in the form of repetitive substructures. In Section 3, we address issues of searching and indexing: we describe there two central tools for these tasks, suffix and subword automata, and consider their implementation issues in massive data contexts. Section 4 deals with basic problems of counting substring statistics and estimating empirical probabilities in early probabilistic models. In Section 5, we address issues of filtering, fingerprinting and related compaction techniques that variously enter data reduction, certification, watermarking, but also approximate patterns comparison and search. In Section 6, we consider problems of compression, mining for associations and other inference issues in strings.

Some preliminary notational conventions follow. Given an alphabet Σ, we use Σ^+ to denote the free semigroup generated by Σ, and set $\Sigma^* = \Sigma^+ \cup \{\lambda\}$, where λ is the empty word. An element of Σ^* is called a *string* or *sequence* or *word*, and is denoted by one of the letters s, u, v, w, x, y and z. The same letters, upper case, are used to denote *random* strings. We write $x = x_1 x_2 ... x_n$ when giving the symbols of x explicitly. The number n of symbols that form x is called the *length* of x and denoted by $|x|$. If $x = vwy$, then w is a *substring* or *factor* of x and the integer $1 + |v|$ is its *(starting) position* in x. Let $I = [i, j]$ be an interval of *positions* of a string x. We say that a substring w of x *begins* in I if I contains the starting position of w, and that it *ends* in I if I contains the position of the last symbol of w.

2. BASIC REGULARITIES AND THEIR DETECTION

It is customary to distinguish among three types of information: syntactic, semantic, and pragmatic, the last one being an attempt to describe the understanding of meaning as a natural process. As much as we would like to get to this third level, it is likely that we shall only be able to occasionally grasp at the second one using tools and methods of the first. In this section, we see that even restricting to syntactic regularities does not make the job trivial.

Syntactic regularities in strings play a pervasive role in many facets of data analysis. Searching for repeated patterns, periodicities, symmetries, cadences, and other similar forms or unusual patterns in objects is a recurrent task in the compression of data, symbolic dynamics, genome studies, intrusion detection, and countless other activities. In many applications, such regularities represent redundancies and, as such, are sought to be removed. This is the case of Data Compression. In textual substitution methods, for example, strings that appear many times in a subject can be economically replaced by pointers to a single common copy. In many other applications, these kind of regularities are sought as carriers of information. This display of duality for information in this context has been known and debated since early years [123, 48].

We concentrate here on a very restricted class of regularities such as cadences, periods, squares, repetitions, palindromes and approximate versions thereof. The first thing to be said is that there are *avoidable* and *unavoidable* such regularities (see, *e.g.*, [31, 99]).

2.1 UNAVOIDABLE REGULARITIES

One remarkable application of the Pigeon Hole Principle leads to establish that if the set of natural numbers $\mathcal{N}$ is partitioned into k classes, then one of the classes contains arbitrarily long arithmetic progressions.

More precisely, we say that the integers $t_1 < t_2 < ... < t_n$ are a *cadence* for word $x_1x_2...x_r$ if $x_{t_1} = x_{t_2} = ... = x_{t_n}$. In this case we also say that n is the *order* of the cadence.

Let now S be a finite subset of $\mathcal{N}$. A cadence of type S is a cadence of the form $\alpha S + \beta$ (*i.e.*, an arithmetic cadence with common difference α when $\alpha, \beta > 0$) For example, ab*b*ab*b*ab*b*aa*b* shows an arithmetic cadence of order 4 with $\alpha = 3, \beta = 3, S = \{1, 2, 3\}$. The following theorems hold (see [99]).

Theorem 1 If A is an alphabet with k letters and n is an integer, there is an integer $N = N(k, n)$ such that every word of length $\geq N$ has an arithmetic cadence of order n

Theorem 2 Let S be any finite subset of $\mathcal{N}$ and A an alphabet with k letters. There exists an integer N depending only of S and k such that every word of length $\geq N$ has a cadence of type S.

2.2 SOME AVOIDABLE REGULARITIES: PERIODS, PALINDROMES AND SQUARES

Periods and periodicities are pervasive notions of string algorithmics. A string z has a *period* w if z is a prefix of w^k for some integer $k > 0$ and some non-empty prefix w of z. Alternatively, a string w is a period of a string z if $z = w^l v$ and v is a possibly empty prefix of w. Often when this causes no confusion, we will use the word "period" also to refer to the length or *size* $|w|$ of a period w of z. A string may have several periods. The shortest period (or period length) of a string z is called *the period* of z. Clearly, a string is always a period of itself. This period is called the trivial period.

A germane notion is that of a border. We say that a non-empty string w is a *border* of a string z if z starts and ends with an occurrence of w. That is, $z = uw$ and $z = wv$ for some possibly empty strings u and v. Clearly, a string is always a border of itself. This border is called the trivial border. The implications of these notions on fast string searching are well understood. In fact, it is not difficult to see that two consecutive occurrences of a word may overlap only if their distance equals one of the periods of w.

A string can have many periods, and corresponding borders. The smallest (resp. longer border) period is *the* period (resp., *the* border) of the string. For example *abaabaababaabaababaabaabaabaab* has borders at *abaabaab*, *abaab* and *ab*.

Once we know how to compute all periods of a string then we also know how to compute all initial palindromes of a string. A *palindrome*

```
procedure maxborder ( y )
    begin
    bord[0] ← −1; r ← −1;
    for m = 1 to h do
        while r ≥ 0 and y_{r+1} ≠ y_m do
            r ← bord[r];
        endwhile
        r = r + 1; bord[m] = r
    endfor
    end
```

Figure 1.1 Computing the longest borders for all prefixes of y

is a string that reads the same forward and backward, *i.e.*, $w = w^R$, where w^R is the *reverse* of string w. For this, we run the algorithm on $w!w^R$ where ! is not in the alphabet. Better palindrome detectors are known. In 1976, G. Manacher showed that all initial palindromes –in fact, all palindromes if one just lets the algorithm go on– of a string can be found in linear time [102].

A string can avoid having any nontrivial period but will not take two periods for long. We give here a weak version of an important result known as the "periodicity lemma" [75, 100].

Lemma 1 If w has two periods of length p and q and $|w|$ is at least $p + q$ then w has period $\gcd(p, q)$.

Proof. Assume w.l.o.g. $p > q$ and consider w_i for arbitrary i. We have that either $i - q \geq 1$ or $i + p \leq n$. In the first case, $x_i = x_{i-q} = x_{i-q+p}$, in the second case $x_i = x_{i+p} = x_{i+p-q}$. Thus, $p - q$ is a period. Repeating the treatment on the pair $p, p - q$ leads to the claim. $\diamond$

The computation of the longest borders (and corresponding periods) of all prefixes of a string is afforded in overall linear time and space. We report one such construction in Figure 1.1, for the convenience of the reader, but refer for details and proofs of linearity to discussions of "failure functions" and related constructs such as found in, *e.g.*, [3, 20, 65].

Once the period structure of the pattern is unveiled, this immediately yields a linear time string searching algorithm. The key element of the algorithm is to maintain, during a text scanning, notion of the longest prefix of the pattern matched so far, and use the border table to jump over intermediate non-viable candidates. These developments will be discussed some more later, in connection with subword automata.

Let $\pi(w)$ denote the shortest non-zero period length of w. A string w such that $|w| \geq 2\pi(w)$ is said to be *periodic*. By the periodicity lemma, in a periodic string w, all periods lengths that are smaller than $|w|/2$, must be multiples of *the period* length $\pi(w)$. A string w such that setting $w = v^k$ implies $k = 1$ is called *primitive*. A *square* is a string w in the form $w = vv$ with v a primitive string. It is natural to wonder whether squares represent avoidable or unavoidable regularities. As is readily seen, on an alphabet of two symbols we can only build a very short string not containing any squares, *i.e.*, a *square-free* string. In fact, in the first three steps we must generate either 010 or 101, at which point adding, say, 0 to 010, introduces the square 00 while adding 1 yields 0101.

At the beginning of the century, A. Thue [128, 129] found that over an alphabet of at least 3 symbols he could build an indefinitely long square-free string. This was achieved by giving a square-free *morphism*, *i.e.*, a rewriting rule that when applied to a square-free string would preserve square-freeness. The morphism considered by Thue is: $rew(a) = abcab$, $rew(b) = acabcb$ and $rew(c) = acbcacb$. Later, S. Istrail (see [34]) gave a more compact morphism that is square free if started on the letter a: $rev(a) = abc$, $rew(b) = ac$, and $rew(c) = b$. As for a binary alphabet, it is possible to show that we can build infinite *cube-free* strings, with obvious meaning.

There are, in principle, about $n^2/2$ possible ways to choose indices i and j for the starting and ending positions of a substring in a string of n symbols, and these might all correspond to distinct strings. Is it possible to have as many squares? As it turns out, there can be only $O(n \log n)$ squares. One way to prove this is by giving an algorithm that enumerates all the squares. M. Crochemore showed in 1981 [57] that this number of squares is also tight: the *Fibonacci* strings, defined by $F_0 = a$, $F_1 = b$, and $F_i = F_{i-1}F_{i-2}$, attain this bound.

There are several efficient or optimal serial [101, 117, 57, 22, 84, 86] and parallel [67, 66, 17, 12] algorithms to test square-freeness and detect all squares. We will discuss some simple criterion and algorithm later.

2.3 QUASIPERIODS AND COVERS

In the Summer of 1990, A. Ehrenfeucht suggested that some repetitive structures defying the classical characterizations of periods and repetitions could be captured by resort to a germane notion of "quasiperiod". In [18] Apostolico and Ehrenfeucht defined *quasiperiodic* strings as *strings which are entirely covered by occurrences of another (shorter) string.* They also gave an $O(n \log^2 n)$ time algorithm to find all maximal quasiperiodic substrings within a given string. Apostolico, Farach and Iliopoulos [19] gave an $O(n)$ time algorithm that finds the quasiperiod of a given string, namely the *shortest string* that covers the string in question. This algorithm was subsequently simplified and improved by Breslauer [44] who gave an $O(n)$ time on-line algorithm, and parallelized by Breslauer [45] and Iliopoulos and Park [90], the latter giving an optimal-speedup $O(\log \log n)$ time parallel CRCW-PRAM algorithm. Moore and Smyth [109] gave an $O(n)$ time algorithm that finds *all strings* that cover a given string. These developments eventually led to the study by Iliopoulos, Moore and Park [89] and by Ben-Amram et al. [32] of covers which are not necessarily aligned with the ends of the string being covered, but are rather allowed to overflow on either side. The sequential algorithm for this problem takes $O(n \log n)$ time [89] and the parallel counterpart [32] achieves an optimal speedup taking $O(\log n)$ time, but using superlinear space.

To understand these developments, it is convenient to modify slightly the notion of a period. A non-empty string u, $|u| \leq |w|$, will be called *a period* of w if w is a substring of u^k, for some integer $k \geq 1$. Clearly, if u is a period of w, then its length $|u|$ is a period length of w, since $|u|$ is a period length of u^k. Moreover, if $u = xy$, then any *rotation* yx of u is also a period of w since $(yx)^{k+1} = y(xy)^k x = yu^k x$ contains w as a substring. A period u of w that is also a prefix of w is called a *left aligned period.* Clearly, given any period length $\pi > 0$ of w, the prefix $w_{[1\ldots\pi]}$ is a left aligned period of w.

A period u is in fact a *regular cover* of w, where occurrences of u appear in w spaced exactly $|u|$ positions apart (other occurrences are also allowed) and the occurrences on the sides can overflow. Given any period u of w, consider the rotation $\hat{u}$ of u such that $\hat{u}$ is also a prefix of w (in other words, $\hat{u}$ is the rotation of u that is a left aligned period of w). If $w = \hat{u}^k$ for some integer k, namely if the regular cover of w by u is also right aligned, then w is said to have an *aligned regular cover* u. If w has no proper aligned regular covers (w itself is always a cover) then w is primitive.

2.3.1 General Covers. One may generalize the notion of a period u that covers w with regular occurrences that are $|u|$ positions apart in w, to covers where the occurrences of u in w are not required to be uniformly spaced, and are allowed, in addition, to overflow on either side. For example, the string $w =$ '*aabaabab*' may be covered by occurrences of $u =$ '*aba*', but the positions of these occurrences in w are not regular and in fact aba is not a period of w. This type of covers were called *general covers* in [89] where a covering string such as our u above is also termed *a seed* of w.

2.3.2 Aligned Covers. Some notable families of covers result by considering covering strings u for w that are not necessarily regularly spaced but are aligned on both sides of w and are not allowed to overflow. Such strings u are said to be aligned covers of w. Given the similarity between non-regular covers and regular covers (periods), aligned covers u of w were named *quasiperiods* of w by Apostolico and Ehrenfeucht [18]. In addition, strings that do not have any non-trivial (shorter) aligned covers were called *superprimitive* and strings that have shorter aligned covers were termed *quasiperiodic*. Observe that any periodic string is also quasiperiodic, but not every quasiperiodic string is periodic. Most of our treatment here is confined to aligned covers.

We describe next few easy facts about periods, borders, and aligned covers.

Lemma 2 If a string z is an aligned cover for a string w then z is a border of w.

Proof. Since the first symbol of w must be covered by z, the string w must start with an occurrence of z. Since the last symbol of w must also be covered by z, the string w must also end with an occurrence of z. That is, z is a border of w. $\diamond$

Note that by this last fact any cover of a string w can be represented by a single integer that is the length of the border of w.

Lemma 3 If a string z covers a string w, then z covers also any possible border v of w such that $|v| \geq |z|$.

Proof. Given any prefix of w, it is covered by z except possibly at most the last $|z| - 1$ symbols of the prefix. Similarly, given any suffix of w, it is covered by z except possibly at most the first $|z| - 1$ symbols of the suffix. Since v is a border of w, it is both a prefix and a suffix, and it must be covered by z. $\diamond$

Lemma 4 Every string has a unique quasiperiod.

Proof. Assume that a string w is covered by two strings u and v, and let w.l.o.g. $|u| \leq |v|$. By Lemma 2, v is a border of w. By Lemma 3, u covers w. Since $u \neq v$, then v is quasiperiodic. $\diamond$

Lemma 5 If a string w has a border z such that $2|z| \geq |w|$, then z covers w.

Proof. z covers the first half of w since it is a prefix of w and the last half of w since it is also a suffix. Therefore, all symbols of w are covered by z. $\diamond$

3. INDEXING, TRANSDUCING AND CHECKING

Various pattern matching techniques and tools (refer, *e.g.*, to [20, 65]) have been developed in the last two decades to detect and count all distinct occurrences of an assigned substring w (the *pattern*) within a longer string x (the *text*). As already mentioned, this problem can be solved in $O(|x|)$ time. In widespread applications, many queries of this kind are performed on a relatively stable repository, and it makes sense to preprocess the text archive so as to get an index on which searches can be carried out in time proportional to the query, rather than archive size. A number of structures achieve this objective, and we describe some of them in this section.

3.1 SUBWORD TREES

Let x be a string of n symbols over the alphabet Σ and \$ an extra character not in Σ. The *expanded suffix tree* T_x associated with x is a digital search tree collecting all non-empty suffixes of $x\$$. Thus, T_x is a tree with n leaves, labeled from 1 to n. Each arc is labeled with a symbol of $\Sigma \cup \{\$\}$. For any i, $1 \leq i \leq n$, the concatenation of the labels on the path from the root of T_x to leaf i is precisely the suffix $suf_i = x_i x_{i+1} ... x_n \$$. Moreover, for any two suffixes suf_i and suf_j of $x\$$, the path associated with their longest common prefix is the same in T_x.

The tree can be interpreted as the state transition diagram of a deterministic finite automaton where all nodes and leaves are final states, the root is the initial state, and the labeled arcs, which are assumed to point downwards, represent part of the state-transition function. The state transitions not specified in the diagram lead to a unique non-final *sink* state. Our automaton recognizes the (finite) language consisting of all substrings of string x. This shows how the tree can be used in an on-line search: given a query pattern y, we follow the downward path in the tree in response to consecutive symbols of y, one symbol at a time. Clearly, y occurs in x if and only if this process takes to a final state. In terms of T_x, we say that the *locus* of a string y is the node α, if it exists, such that the path from the root of T_x to α is labeled y. Thus, a string y occurs in x if and only if y has a locus in T_x. Finding this out takes $O(t \cdot |y|)$ character comparisons, where t is the time necessary to traverse a node, which is constant for a finite alphabet. Note that this only answers whether or not y occurs in x. However, one can easily prove the following

Lemma 6 If y has a locus α in T_x, then the occurrences of y in x are all and only the labels of the leaves in the subtree of T_x rooted at α.

Thus, if we wanted to know where y occurs, it would suffice to visit the subtree of T_x rooted at node α, where α is the node such that the path from the root of T_x to α is labeled y. Such a visit requires time proportional to the number of nodes encountered, and the latter can be $\Theta(n^2)$ on the expanded suffix tree. This is as bad as running an offline search naively, but we will see shortly that a much better bound is possible.

An algorithm for the construction of the expanded T_x is readily organized. We start with an empty tree and add to it the suffixes of $x\$$ one at a time. Conceptually, the insertion of suffix suf_i $(i = 1, 2, ..., n)$ consists of two phases. In the first phase, we search for suf_i in T_{i-1}. Note that the presence of \$ guarantees that every suffix will end in a distinct leaf. Therefore, this search will end with failure sooner or later. At that point, though, we will have identified the longest prefix of suf_i that has a locus in T_{i-1}. Let $head_i$ be this prefix and α the locus of $head_i$. We can write $suf_i = head_i \cdot tail_i$ with $tail_i$ nonempty. In the second phase, we need to add to T_{i-1} a path leaving node α and labeled $tail_i$. This achieves the transformation of T_{i-1} into T_i. It is clear that this construction takes time $\Theta(n^2)$ and $O(n^2)$ space.

It is instructive to examine the cost of this procedure in terms of the two main phases of each suffix insertion. If the symbols of x are all different, then T_x contains $\Theta(n^2)$ arcs. Finding the (empty) head only charges linear time overall, and the heaviest charges come from adding the tail paths. At the other extreme, consider $x = a^{n-1}$. In this case, tail paths charge linear time overall and the quadratic work is done in order to find the heads.

It is easy to reduce the work charged by tails by resorting to a more compact representation of T_x. Specifically, we collapse every chain formed by nodes with only one child into a single arc, and label that arc with a substring, rather than with a symbol of $x\$$. Such a *compact* version of T_x has at most n internal nodes, since there are $n + 1$ leaves in total and now every internal node is branching. Clearly, it takes little to adapt the details of the direct construction.

With the new convention, the tree for a string formed by all different symbols only requires 1 internal node, namely, the root. Except for arc-labeling, the construction of such a tree is performed in linear time, since adding a path takes now constant time per suffix. However, there is no improvement in the management of the case $x = a^{n-1}$, in which finding the heads still requires $\Theta(n^2)$ time.

While the topology of the tree requires now only $O(n)$ nodes and arcs, each arc is labeled with a substring of $x\$$. We have seen that the lengths of these labels may be $\Theta(n^2)$ (think again of the tree for a string formed by all different symbols). Thus, as long as this labeling policy is maintained, T_x will require $\Theta(n^2)$ space in the worst case, and it is clearly impossible to build a structure requiring quadratic space in less that quadratic worst-case time. Fortunately, a more efficient labeling is possible which allows us to store T_x in linear space. For this, it is sufficient to encode each arc label into a suitable pair of pointers in the form $[i, j]$ to a single common copy of x. For instance, pointer i denotes the starting position of the label and j the end. Now T_x takes linear space and it makes sense to investigate its construction in better than quadratic time.

As already seen, the time consuming component of suffix insertion is in finding the heads. For every i, this phase starts at the root of T_{i-1} and essentially locates the longest prefix $head_i$ of suf_i that is also a prefix of suf_j for some $j < i$. Note that $head_i$ will no longer necessarily end at a node of T_{i-1}. When it does, we say that $head_i$ has a *proper locus* in T_{i-1}. If $head_i$ ends inside an arc leading from some node α to some node β, we call α the *contracted locus* and β the *extended locus* of $head_i$. We use the word *locus* to refer to the proper or extended locus, according to the case. It is trivial to upgrade head location in such a way that the procedure creates the proper locus of $head_i$ whenever such a locus does not already exist. Note that this part of the procedure only requires constant time.

The above discussion embodies the obvious principle that the construction of a digital search tree for an arbitrary set of words $\{w_1, w_2, ..., w_k\}$ cannot be done in time better than the $\sum_{i=1}^{k} |w_i|$ in the worst case. This seems to rule out a better-than-quadratic construction for T_x, even when the tree itself is in compact form. However, the words stored in T_x are not unrelated, since they are all suffixes of a same string. In fact, clever constructions, such as in [108, 130, 135] are available that build the tree in time $O(n \log |\Sigma|)$ and linear space. One key element in such constructions is offered by the following easy facts.

Lemma 7 If $w = av$, $a \in \Sigma$, has a proper locus in T_x, then so does v.

Lemma 8 For any i, $1 \leq i \leq n$, $|head_{i+1}| \geq |head_i| - 1$.

Proof. Assume the contrary, *i.e.*, $|head_{i+1}| < |head_i| - 1$. Then, $head_{i+1}$ is a substring of $head_i$. By definition, $head_i$ is the longest prefix of suf_i that has another occurrence at some position $j < i$. Let $x_j x_{j+1} ... x_{j+|head_i|-1}$ be such an occurrence. Clearly, any substring of $head_i$ has an occurrence in $x_j x_{j+1} ... x_{j+|head_i|-1}$. In particular, $x_{j+1} x_{j+2} ... x_{j+|head_i|-1} = x_{i+1} x_{i+2} ... x_{i+|head_i|-1}$, hence $x_{i+1} x_{i+2} ... x_{i+|head_i|-1}$ must be a prefix of $head_{i+1}$. $\diamond$

To exploit this fact, *suffix links* are maintained in the tree that lead from the locus of each string av to the locus of its suffix v. Here we are interested in Lemma 7 only for future reference.

Using these tools, McCreight proved the following.

Theorem 3 The suffix tree in compact form for a string of n symbols can be built in $O(t \cdot n)$ time and $O(n)$ space, where t is the time needed to traverse a node.

As the discussion unravels, we shall see several applications of suffix trees and their companion structures, ranging from the detection of regularities, string statistics of various kinds, finding common subwords in words, etc.

When it comes to the actual allocation in memory of a suffix tree, one faces a number of design choices, prominent among which those pertaining to the implementation of nodes. There are three main possibilities in this regard. The first one is to implement each node as an array of size $|\Sigma|$. This yields fast searches, but is likely to introduce an unbearable amount of waste even for small alphabets. The second option is to store each node as a linked list (or, better, as a balanced search tree). This keeps space to a minimum, but introduces an overhead on the search. Finally, one may implement the adjacency of a node as part of a global hash coding. This yields expected constant time search within overall $\Theta(n \log n)$ space.

In massive applications, even linear space can be problematic: at 20 bytes per node and with a number of nodes 1.5 times the number of symbols in the input string, a text of size n needs approximately $30n$ bytes of storage space. In general, although the size of the suffix tree depends on the particular implementation, one might expect it to be never lower than 20 bytes per input symbol (or *bps*) in the worst case. We refer to [95] for a comparative study of various space-efficient allocations. Other alternatives have been studied more recently, specially in connection with secondary memory or external storage, resulting in variants called *blind tries* (see, *e.g.*, [72] and references therein).

A space-efficient alternative to a suffix tree is offered by the *suffix array* [105]. This is a essentially a table of the suffixes sorted in lexicographic order, plus some auxiliary information. The implied query technique is inspired by binary search. Specifically, the suffix array of the text x is the structure composed of the two tables POS and LCP. Table POS satisfies the condition:

$$x[POS[1]...n] < x[POS[2]...n] < \ldots < x[POS[n]...n].$$

The second table LCP contains the prefixes common to consecutive suffixes, *i.e.*, LCP is defined, for $i = 1, \ldots, n-1$, by

$$LCP[i] = |lcp(x[POS[i]...n], x[POS[i+1]...n])|,$$

where lcp denotes the length of the longest common prefix of two words.

The preparation of the text, lexicographical ordering of its suffixes and common prefix calculations, can be carried out in time $O(n \log n)$. The reader is encouraged to obtain this by post-processing of a suffix tree. This resulting structure can be shown to support $O(m + \log n)$ time search for a pattern of length m in the text. Note that ordinary binary search would achieve only $O(m \times \log n)$ time, but the technique in [105] uses combinatorial properties of the longest common prefixes to reduce the number of symbol comparisons and the total running time.

In many applications, notably, in data compression, suffix trees and arrays have to be built repeatedly. This exacts a considerable toll irrespective of the method adopted. Ideally, one would like to build the tree once and then maintain it, together with updated annotations of various nature, following every substring selection and removal. Linear time algorithms for dynamically maintaining the tree under deletion of a string were originally proposed by McCreight together with his construction. Similar problems have been studied by Fiala and Green [73] in the context of sliding window compression. More recently, Larsson [96] showed that Ukkonen's algorithm can be easily extended to accommodate the sliding window update of the suffix tree in amortized linear time. Gu *et al.* [85] introduced a new data structure for dynamic text indexing that supports insertion and deletion of a single character in $O(\log n)$ time and the i updates involving a substring w that occurs occ_w times in $O(|w| + occ_w \log i + i \log |w|)$. Additional recent efforts and references addressing the dynamic maintenance of various indices are found in [72].

3.2 SUBWORD AUTOMATA AND FACTOR TRANSDUCERS

An important companion to the suffix trees and arrays is the *directed acyclic word graph (DAWG)*, a data structure specifically designed to represent the set *Fac(x)* of all substrings of a word. Roughly, the graph D_x, called the *suffix dawg* of x, may be obtained by identifying first and then superimposing isomorphic subtrees of the uncompacted tree T_x. An advantage of dawgs is that each edge is labeled by a single symbol, and they are somehow more convenient to use whenever information is associated with edges rather than with nodes. We consider only dawgs representing the set *Fac(x)*, but it is clear that the analog structures could be built for other sets of words, *e.g.*, the set of subsequences of a string.

The possible applications of suffix dawgs are essentially the same as those of suffix trees. Indexing is the main purpose of these data structures. Below, we demonstrate the use of the suffix dawg of a pattern to speed up its search in a text.

A node in the graph D_x naturally corresponds to a set of substrings of the text, namely, substrings having the same right context. It is not difficult to be convinced that all these substrings have the following property: their first occurrences end at the same position in the text. The converse is not necessarily true, but this characterization gives an intuition of the definitions that follow.

Let z be a substring of x, and let $endpos(z)$ denote the set of all positions in x where an occurrence of z ends. Let y be another substring of x. Clearly, the subtrees of T_x rooted at z and y are isomorphic iff $endpos(x) = endpos(y)$ (recall that x is completed by a special end marker). In the graph D_x, paths relative to substrings with the same *endpos* sets end on a same node. The small size of dawgs is due to the special structure of the family of sets *endpos*. We associate each node ν of the dawg with the length $val(\nu)$ of the longest word leading to it from the source. The nodes of the dawg are, in fact, equivalence classes of nodes of the uncompacted tree T_x under subtree isomorphism. In this sense, $val(\nu)$ is the longest representative of its equivalence class. One could also regard the nodes of the dawg as equivalence classes of substrings of the text, since the nodes in T_x are in one-to-one correspondence with the distinct substrings of x.

The notion of border and failure function has an exact counterpart in dawgs. Let ν be a node of D_x distinct from the source node. We define $suf[\nu]$ as the node μ such that $val(\mu)$ is the longest suffix of $val(\mu)$ not equivalent to it, *i.e.*, corresponding to a node other than μ. By convention, the *suf* of the source is the source itself. The table *suf* is analogous to the table *bord* defined earlier. The links implied by the table *suf* are called *suffix links*. These links connect the nodes in a tree structure, whereby $suf[\mu]$ is interpreted as the father of μ. It so happens that this tree embodies the containment relation of the *endpos* sets.

For any string x with n symbols over an arbitrary alphabet, D_x has a number of nodes $N \leq 2n$, and a number of edges $E < N + n - 1$.

Proof. The main property used here is that for any two *endpos* sets, these are either disjoint or one is contained in the other. Thus, the family of *endpos* sets has a tree structure. All leaves are pairwise disjoint subsets of $\{1, 2, ..., n\}$. Hence, there are at most n leaves. We partition the nodes into two (disjoint) subsets according to whether or not $val(\mu)$ is a prefix of x. The number of nodes in the first subset is exactly $n+1$, the number of prefixes of x. We now count the number of nodes in the second subset. Let ν be a node such that $val(\nu)$ is not a prefix of x. Then $val(\nu)$ occurs in at least two different right contexts in x, whence there are at least two nodes μ and μ' (corresponding to two different factors of x) are such that $suf[\mu] = suf[\mu'] = \nu$. Hence μ has at least two children in the tree induced by suf. Since the tree has at most n leaves (corresponding to non-empty prefixes), the number of its nodes is smaller than n. In conclusion, D_x contains at most $(n+1)+(n-1) = 2n$ nodes.

To bound the number of edges, consider a spanning tree T over D_x, and count separately the edges in the tree and those outside. Since there are N nodes in the tree, this accounts for $N - 1$ edges. Let us count the other edges of D_x. With each such edge (ν, μ), we associate the suffix zay of x such that z is the label of the path in T going from the source to ν, a is the label on (ν, μ), y is the substring extending za into a suffix of x. It is clear that this correspondence is one-to-one with the suffixes. Moreover, the empty suffix is not considered, nor is x itself because it is already in the tree. This leaves $n - 1$ suffixes, which is the maximum number of edges outside T, whence the number of edges in D_x cannot exceed $N + n - 1$. $\diamond$

Although the size of D_x is linear, it is not always strictly minimal. If minimality is understood in the sense of finite automata, *i.e.*, restricted to the number of nodes, then D_x is the minimal automaton for the set of suffixes of x. The minimal automaton for $Fac(x)$ can be, indeed, slightly smaller.

A simple construction of dawgs can be based on a transformation of the suffix tree. The reader is referred to [37] for an on line construction. The basic procedure is the computation of the equivalence classes associated with subtrees. This is based on a classical algorithm for tree isomorphism. Here we just recall the final result without proof.

Lemma 9 Let T be a rooted ordered tree in which the edges are labeled by symbols from a finite alphabet. Then, isomorphic classes of all subtrees of T can be computed in linear time.

An application of Lemma 9 provides a linear time transformation of T_x into a compact version of D_x, in which edges are labeled by words and no node has only one outgoing edge. From this, the transition to the final dawg is easy. Informally, the first step is to juxtapose nodes of T_x that are roots of isomorphic subtrees. The resulting structure differs from a dawg in that edges are labeled by strings rather than symbols. Breaking down each edge risks introducing a quadratic number of nodes. The following approach preserves the linearity of space. Let the *weight* of an edge be the length of its label, and let $inedge(\nu)$ be the heaviest incoming edge for ν and z the corresponding label. Break down $inedge(v)$ its label into consecutive unit edges. At this point, for each one of the other incoming edges of ν with a label az can be implemented by directing a new edge, labeled by the symbol a, to the node μ, on the chain now replacing *inedge*, such that the path from μ to ν is labeled by z. It is crucial that all these local transformations can be performed on all nodes ν independently.

This algorithm cannot be used directly to build the smallest automaton accepting $Fac(x)$. The on-line construction of these [59] is more technical than that of suffix dawgs given in [37, 60].

There is a very close relationship between our two "good" representations for the set $Fac(x)$. For this discussion, we assume that the string x starts with a symbol occurring only at the beginning of x. In this case the relation-ship between dawgs and suffix trees is particularly tight and simple.

Lemma 10 Assume that x has a unique left most symbol. Then the following three properties are equivalent:

$endpos(w) = endpos(y)$ in x;
$first\text{-}pos(w^R) = first\text{-}pos(y^R)$ in x^R;
w^R, y^R are contained in the same chain of the uncompacted T_{x^R}.

It follows as a corollary that the reversed val's of nodes of the suffix tree T_{x^R} are the longest representatives of equivalence classes of substrings of x. Hence the nodes of T_{x^R} coincide with those of D_x. In T_x, define the *shortest extension* link $sext[a,\nu]$ as the node μ such that $y = val(\mu)$ is the shortest word having prefix az, where $z = val(\nu)$. If there is no such node μ, then $sext[a,\nu] = nil$. Observe the relationship between sext links and suffix links. The following properties hold.

Theorem 5 If x has the unique left most symbol then D_x coincides with the graph of sext links of $T = T_{x^R}$.

Theorem 6 If x starts with a unique symbol then the tree of suffix links of D_x coincides with the suffix tree T_{x^R}.

Like trees, huge dawgs, such as arising in the design of thesaura for language and speech applications, present considerable problems of efficient storage and access. Compressed versions exist that variously expose and exploit the relationship between D_x and D_{x^R}. It is also possible to make a symmetric version of a dawg, *i.e.*, data structures that represent simultaneously $Fac(x)$ and $Fac(x^R)$.

A simple application of either suffix trees or subword dawgs is to compute a longest common substring of two strings. With trees, this is the deepest common node in the intersection of the two trees. With dawgs, one may build on line the dawg of the shortest word and then travel on this with the longer one. The overall algorithm results in an approach to string-matching further highlighted below. The reader is encouraged to work out the details.

First, build the dawg D_y of the pattern. The text x is scanned then from left to right. At some generic step, letting w be the prefix of x that has just been processed, we maintain that we know the longest suffix s of w that is a substring of y. We want now to compute the same value associated with the next prefix wa of x. It is clear that D_y yields this value immediately via forward transitions whenever sa is a substring of y. If this is not the case, the next state in the dawg is reached via suffix links. This is similar to using the links subtended by the computation of borders. Each transition on a forward link corresponds to advancing on x by one character, whereas transitions on suffix links represent forward shifts for the pattern relative to the text. Either action cannot be performed more than n times in total, whence the overall linear time bound.

3.3 SEARCHING FOR WORD SETS AND REGULAR EXPRESSIONS

The most general exact searching problem may be cast in terms of regular expressions. Regular expressions describe sets of strings resulting by a finite number of concatenations $(\cdot)$, union $(+)$ and *star* operator $(*)$ to the symbols of an alphabet, where the $*$ operator is the reflexive transitive closure of concatenation. For instance, $(0\cdot 1)^* + (0\cdot 1\cdot 1)^*$ is the set of all strings in either one of the forms 01010101... or 011011011011.... The problem is, given a regular expression $\mathcal{E}$, to preprocess it in order to locate all occurrences of words of the associated language $lang(\mathcal{E})$ that occur in any given word x.

The special case where $\mathcal{E}$ is a finite set of words is efficiently handled by suitable extensions of the dawg and its companion structures. The classical solution to the general problem is composed of two phases. First, transform the regular expression $\mathcal{E}$ into a nondeterministic automaton that recognizes the language described by $\mathcal{E}$, following a construction due to Thompson (refer to, e.g., [65]). Second, simulate the obtained automaton with input word y in such a way that it recognizes each prefix of y that belongs to $\Sigma^* \cdot lang(\mathcal{E})$. Both phases are linear in the input. In particular, a nondeterministic automaton taking space linear in the length of the regular expression is easily built by iterated serial/parallel composition of smaller automata over the alphabet $\Sigma \cup \{\lambda\}$, using transitions on the empty symbol λ as connectors. Composition of constituent automata under each of the operations induced by $+$, $\cdot$ or * can be implemented to work in constant time. Combined with a prudent parsing of $\mathcal{E}$ this leads to the following result:

Theorem 7 Let $\mathcal{E}$ be a regular expression. The nondeterministic automaton recognizing $lang(\mathcal{E})$ can be computed and stored in time and space $O(|\mathcal{E}|)$.

The derivation of such an automaton proves one half of a central theorem of Kleene, which sets the equivalence between the languages recognized by finite automata and those described by a regular expression.

Theorem 8 (Kleene, 1956) A language is recognized by a finite automaton if and only if it is can be described by a regular expression.

However, it is well known that the transformation of a nondeterministic automaton into a deterministic one is accompanied by an exponential explosion in the number of states. This poses a problem in the searches, since the search for end-positions of words in $lang(\mathcal{E})$ is performed by a simulation of a deterministic automaton recognizing $\Sigma^* lang(\mathcal{E})$. To circumvent this, the determinization is just simulated at search time: at any given time during the search, the automaton will not be in a single state, but rather in a set of states, the search itself taking care of dynamically maintaining knowledge of this set. A central notion for this process, related to λ-transitions, is that of λ-*closure* for a set of states S. This is the set of states Q reachable from Q solely through λ-transitions. Once the closure of a set of states is known, it is possible to compute effectively the transitions induced by any input symbol.

The simulation of a regular-expression-matching automaton consists in repeating the two operations "closure" and "transitions on a set of states". With careful implementation, based on standard manipulation of sets and queues, the time and the space required to perform either part is linear in the size of sets of states involved. This leads to the following

Theorem 9 Given a regular expression $\mathcal{E}$, testing whether a word y belongs to $lang(\mathcal{E})$ can be done in time $O(|\mathcal{E}| \times |y|)$ and space $O(|\mathcal{E}|)$.

Note that the original problem is different, in that it requires that the answer to the test be reported for each substring of the text x, and not only on x itself. But no transformation of the automaton for $lang(\mathcal{E})$ is necessary. A mere transformation of the search phase of the algorithm is sufficient: at each iteration of the closure computation, the initial state is integrated to the current set of states. By doing so, each substring of x is tested, and the following is established.

Theorem 10 Let $\mathcal{E}$ be a regular expression and x be a word. Finding all end-positions of subwords of x that are recognized by the automaton associated with $lang(\mathcal{E})$ can be performed in time $O(|\mathcal{E}|\,|x|)$ and space $O(|\mathcal{E}|)$. The time spent on each symbol of x is $O(|\mathcal{E}|)$.

As mentioned, the drawback of performing regular-expression-matching by deterministic automata is that the automaton can have a number of states exponential in the length of $\mathcal{E}$. This is the situation, for example, when

$$\mathcal{E} = \mathsf{a}\,\overbrace{(\mathsf{a}+\mathsf{b})\cdots(\mathsf{a}+\mathsf{b})}^{m-1 \text{ times}}$$

for some $m \geq 1$; here, the minimal deterministic automaton recognizing $\Sigma^* lang(\mathcal{E})$ has exactly 2^m states since the recognition process has to memorize the last m symbols read from the input word x. However, not all states of the deterministic automaton for $\Sigma^* lang(\mathcal{E})$ are necessarily met during the search phase. This suggests a lazy construction of the deterministic automaton during the search as a possible practical alternative, which is effectively implemented in efficient software.

4. MODELING, COUNTING, ESTIMATING AND SCORING

In many applications, repetitions of substrings and other substructures represent redundancies and, as such, may be sought just so as to be removed. This is the case of Data Compression. In textual substitution methods, for example, strings that appear many times in a subject can be economically replaced by pointers to a single common copy. In many other applications, these same kinds of regularities are sought as carriers of information. In applications ranging from Consumer Prediction to Data Mining, Intrusion Detection and Security, Protein and other Biological Sequence Classification, the idea is to infer a consistent behavior from some protocol of past records and then use it to predict future behavior or detect malicious practices. This entails some notion of sequence similarity, whereby having established some set of behavioral sequences as constituting the normal profile, any new sequence can be compared to the dictionary and possibly classified or spotted as abnormal. Learning takes place in general from both positive and negative samples.

As already mentioned, this display of somewhat of a duality for the notion of information has been sensed and debated for decades [123, 48]. In Shannon's terms, for instance, the self-information of string x relative to a given source P is measured by $-\log P(x)$. This notion is central to coding: the mean codelength of any Uniquely Decipherable Code for strings of the same length is lower bounded by the entropy, the mean of self information. For Brillouin, information is related to redundancy and negentropy, entropy is chaos.

Either way, in our applications we do not know the source probabilities, which are in fact fictitious entities or models. One pervasive problem is therefore to estimate the probabilities from the observed strings, to be used in the design of codes for compression or other purposes. The domains in which this need arises are countless: Prediction, Inference, Modeling, Learning, and Universal Coding, to quote a few. From an information theoretic standpoint, an important question is how to define a notion of information relative to a class of sources. From the standpoint of Pattern Matching, interesting questions revolve around how computationally expensive it is to estimate probabilities and related deviations within a given class. Below, we consider some preliminary counts and statistical computations. Later in this chapter, we will also consider issues of modeling by Markov Chains and related Finite State Automata sources.

4.1 BASIC STRING COUNTS AND STATISTICS

The tree T_x is a remarkable compendium of the structure of a string. It can be immediately adapted to solve problems such as finding the longest repeated substring, the longest substring common to many strings, or finding squares or palindromes, etc. To find squares, for instance, it suffices to note the following:

Lemma 11 There is a square in x iff there is a node μ in T_x such that the subtree rooted at μ contains two consecutive leaves i and j such $j - i \leq |w(\mu)|$, where $w(\mu)$ denotes the word on the path from the root to μ.

In fact, if $j - i \leq |w(\mu)|$ as stated, then the two occurrences of $w(\mu)$ at i and j are adjacent or overlap, whence we must have a square. We leave it for the reader to show that in the converse of the proof leaves i and j are indeed consecutive as claimed. The reader might also find it interesting to derive a similar criterion for the detection of palindromes on the tree of $x\$x^R$.

Also the count of occurrences of all substrings of a string x is an easy application of T_x. The number of occurrences (with overlap) of a string w of x is trivially given by the number of leaves reachable from the node closest to the locus of w in T_x, and this is irrespective of whether or not w ends in the middle of an arc. Thus, labeling every internal node α of T_x with the number $c(\alpha)$ of the leaves in the subtree rooted at α yields this statistics for all substrings of x.

The problem becomes more involved if we wanted to build a similar index for the statistics without overlap, in which we count, for each substring, its maximum number of nonoverlapping occurrences. It is seen that this transition induces a twofold change in the structure: on the one hand, the weight in each node does no longer necessarily coincide with the number of leaves; on the other, extra nodes must be introduced to account for changes in the statistics that occur in the middle of arcs. The efficient construction of this augmented index in minimal form (*i.e.*, with the minimum possible number of unary nodes) is quite elaborate [23]. For a string x, the resulting structure is denoted $\hat{T}(x)$ and called the *Minimal Augmented Suffix Tree* of x. It is not difficult to build $\hat{T}_x$ in $O(n^2)$ time and space by embedding the necessary weighting as part of the iterated suffix insertion procedure, hence at an expected cost of $O(n \log n)$ [24]. The time required by the construction given in [23] is instead $O(n \log^2 n)$ in the worst case. The number of auxiliary nodes can be bounded by $O(n \log n)$, but it is not clear that such a bound is tight.

Consider for a moment the problem of defining and computing empirical probabilities. One problem here is that the notion of empirical probability is not straightforward. Fortunately, empirical *conditional* probabilities often turn out to be less controversial. One ingredient in the computation of empirical probabilities is the count of occurrences of a string in another string or set of strings. We concentrate on this problem first. Since there can be $O(n^2)$ distinct substrings in a string of

n symbols, a table storing the number of occurrences of all substrings of the string might take up in principle $\Theta(n^2)$ space. However, we just saw that linear time and space suffice to build an index suitable to return, for any string w, its χ_w count in x. Here we want to analyze this fact a little more closely. We begin by formulating a "left-context" property, symmetric to one already seen, and conveniently adapted from [37].

Given two words x and y, let the *start-set* of y in x be the set of *occurrences* of y in x, *i.e.*, $pos_x(y) = \{i : y = x_i ... x_j\}$ for some i and j, $1 \leq i \leq j \leq n$. Two strings y and z are equivalent on x if $pos_x(y) = pos_x(z)$. The equivalence relation instituted in this way is denoted by $\equiv_x$ and partitions the set of all strings over Σ into equivalence classes. Recall that the *index* of an equivalence relation is the number of equivalence classes in it.

Lemma 12 The index k of the equivalence relation $\equiv_x$ obeys $k < 2n$.

Lemma 12 is established in analogy to its right-context counterpart seen in connection with dawgs. In the example of the string *abaababaabaababaababa*, for instance, $\{ab, aba\}$ forms one such C_i-class and so does $\{abaa, abaab, abaaba\}$. Lemma 12 suggests that we might only need to compute empirical probabilities for $O(n)$ substrings in a string with n symbols. The considerations developed earlier make this statement more precise and in fact give one possible proof of it.

We are now ready to consider more carefully the notion of empirical probability. One way to define the empirical probability of w in x is to take the ratio of the count of the number χ_w to $|x| - |w| + 1$, where the latter is interpreted as the maximum number of possible starting positions for w in x. For w and v much shorter than x we have that the difference between $|x| - |w| + 1$ and $|x| - |wv| + 1$ is negligible, which means that the probabilities computed in this way and relative to words that end in the middle of an arc do not change, *i.e.*, we only need to compute those associated with strings that end at a node of the compact T_x.

This notion of empirical probability, however, assumes that every position of x compatible with w length-wise is an equally likely candidate. This is not the case in general, since the maximum number of possible occurrences of one string within another string depends crucially on the compatibility of self-overlaps. For example, the pattern *aba* could occur at most once every two positions in *any* text, *abaab* once every three, etc. Compatible self-overlaps for a string z depend on the structure of the periods of z. An alternative count can be defined as follows.

Definition The maximum possible number of occurrences of a string w into another string x is equal to $(|x| - |w| + 1)/|u|$, where u is the smallest period of w.

According to this definition, in order to compute the empirical probabilities of, say, all prefixes of a string we need to know the borders or periods of all those prefixes. In fact, we know we can manage to carry out *all* the updates relative to the set of prefixes of a same string in *overall* linear time, thus in amortized constant time per update.

The construction of Fig. 1.1 may be applied, in particular, to each suffix suf_i of a string x while that suffix is being inserted as part of the direct tree construction. This would result in an annotated version of T_x in overall quadratic time and space in the worst case. Note that, unlike in the case of empirical probabilities previously considered, the period —and thus also the empirical probabilities according to our definition above— may change along an arc of T_x, so that we may need to compute explicitly all $\Theta(n^2)$ of them. However, if we were interested in such probabilities only at the nodes of the tree, then these could still be computed in overall linear time. The key to this latter fact is to run a suitably adapted version of `maxborder` walking on suffix links "backward", *i.e.*, traversing them in their reverse direction, beginning at the root of T_x and then going deeper and deeper into the tree. One way to visualize this process is as follows. Imagine first the "co-tree" of T_x formed by the reversed suffix links: we can visit such a structure depth first and simultaneously run a procedure much similar to `maxborder` to assign periods to all nodes of T_x. Correctness rests on the fact that for any word w the periods of w and w^R coincide. We shall see shortly that in situations of interest to us we can limit computation to the nodes of T_x.

Lemma 13 The set of empirical probabilities of all (short) words of x that have a proper locus in T_x can be computed in linear time and space.

Consider now conditional empirical probabilities, defined as the ratio between the observed occurrences of $s\sigma$ and the occurrences of $s*$, denoting string s followed by any other symbol. The first thing to observe is that the value of this ratio persists along each arc of the tree, *i.e.*,

$$\tilde{P}(\sigma|s) = \chi_s/\chi_{s\sigma} = 1$$

for any word s ending in the middle of an arc of T_x and followed there by a symbol σ.

Let ν' be the locus of string s'. Recall that $sext[\nu', \sigma]$ is the node ν which is the locus of the shortest extension of as' having a proper locus in T_x. Setting *sext* links is an easy linear post-processing of T_x. Along these lines, attaching empirical conditional probabilities only to the branching nodes of T_x is doable and suffices. As there are $O(n)$ such nodes, and the alphabet is finite, the collection of all conditional probability vectors for *all* subwords of x takes only linear space.

Lemma 14 The set of empirical conditional probabilities of all (short) words of a string x over a finite alphabet can be computed in linear time and space.

An important class of applications, which includes some core tasks of molecular sequence analysis and information retrieval, involves counting, estimating and comparing to expectation not the occurrence number of a word in a text but rather the number of sequences in a given family that contains that word. With some provisos, the constructions just highlighted may be adapted to deal with this notion. The reader is encouraged to develop the details.

4.2 GLOBAL DETECTORS OF UNUSUAL WORDS

As mentioned, the identification of strings that are, by some measure, redundant or rare in the context of larger sequences is variously pursued in order to compress data, unveil structure, infer minimal or compact descriptions, and for purposes of feature extraction and classification. Once a statistical index is built and empirical probabilities are computed, the next step is thus to annotate it with the expected values and variances and measures of discrepancy thereof, under some adopted probabilistic model. This may be still rather bulky in practice. For a given probabilistic model and measure of departure from expected frequency, it is possible to come up with an "observed" string such that all of its $\Theta(n^2)$ substrings are surprisingly over- or under-represented. This means that a table of the "surprising" substrings of a string can contain in principle a number of entries quadratic in the length of that string. As it turns out, it is possible to show that under several accepted measures of frequency deviation, the candidates over- or under-represented words are restricted to the $O(n)$ words that end at internal nodes of a compact suffix tree. as opposed to the $\Theta(n^2)$ possible substrings. Combined with some of the constructions discussed earlier in this section, this leads to the design of global detectors for unusual words that take linear space and linear time to build [15].

To make our discussion more precise, we need to agree on some measure of "surprise". Perhaps the naivest possible measure is the difference:

$\delta_w = f_w - (n-|w|+1)\hat{p}$, where $\hat{p}$ is the product of symbol probabilities for w and $Z|w$ takes the value f_w. Let us say that an over-represented (respectively, under-represented) word w in some class C is δ-significant if no extension (respectively, prefix) of w in C achieves at least the same value of $|\delta|$.

Theorem 11 *The only over-represented δ-significant words in x are the $O(n)$ ones that have a locus in T_x. The only under-represented δ-significant words are the ones that represent one unit-symbol extensions of words that have a locus in T_x.*

Proof. We prove first that no over-represented δ-significant word of x may end in the middle of an arc of T_x. Specifically, any over-represented δ-significant word in x has a proper locus in T_x. Assume for a contradiction that w is a δ-significant over-represented word of x ending in the middle of an arc of T_x. Let $z = wv$ be the shortest extension of w with a defined locus in T_x, and let $\hat{q}$ be the probability associated with v. Then, $\delta_z = f_z - (n - |z| + 1)\hat{p}\hat{q} = f_z - (n - |w| - |v| + 1)\hat{p}\hat{q}$. But we have, by construction, that $f_z = f_w$. Moreover, $\hat{p}\hat{q} < \hat{p}$, and $(n - |w| - |v| + 1) < (n - |w| + 1)$. Thus, $\delta_z > \delta_w$. For this specification of δ, it is easy to prove symmetrically that the only candidates for δ-significant under-represented words are the words ending precisely one symbol past a node of T_x. $\diamond$

It is possible to prove similar properties for more sophisticated measures of surprise characterized by definitions of δ of the more general form: $\delta_w = (f_w - E_w)/N_w$, where: (a) f_w is the frequency or count of the number of times that the word w appears in the text; (b) E_w is the typical or average nonegative value for f_w (and E is often chosen to be the expected value of the count); (c) N_w is a nonnegative normalizing factor for the difference. (The N is often chosen to be the standard deviation for the count.)

Once one is restricted to the branching nodes of T_x or their one-symbol extensions, it becomes even possible to compute all typical count values E (usually expectation) and their normalizing factors N (usually standard deviation) and other measures discussed earlier in overall linear time and space. For strings emitted by a source with i.i.d. symbols, this is easy to see for expectations but becomes more complicated with variances. To see this, let x be the observed string and $y = y_1 y_2 \ldots y_m$ $(m < (n+1)/2)$ be an arbitrary but fixed pattern. For $i \in \{1, 2, \ldots, n - m + 1\}$, define $Z_i|y$ to be 1 if y occurs in X starting at position i and 0 otherwise. We are interested in the expected value and variance of $Z|y$, the total number of occurrences of y in X:

$$Z|y = \sum_{i=1}^{n-m+1} Z_i|y.$$

It is immediate that $E[Z|y] = (n-m+1)\hat{p}$, where, with p_i denoting the probability for any given k that $X_k = y_i$, $\hat{p} = \Pi_{i=1}^{m} p_i$.

For any symbol a in Σ, computing the expected value $Z|ya$ from $\hat{p}$ and the probability of a is trivially done in constant time. Thus, the expected values associated with all prefixes of a string can be computed in linear time. Attaching these values to the nodes of T_x is easily accomplished in linear time by walking backward on suffix links.

For $m \leq (n+1)/2$, it is possible to express the variance in the following form (the case $m > (n+1)/2$ is quite similar) [15]:

$$Var(Z|y) = (n-m+1)\hat{p}(1-\hat{p}) - \hat{p}^2(2n-3m+2)(m-1)$$
$$+2\hat{p}\sum_{l=1}^{s_m}(n-m+1-d_l)\Pi_{j=m-d_l+1}^{m} p_j$$

where the d_l's are the *periods* of y that satisfy

$$1 \leq d_1 < d_2 < ... < d_{s_m} \leq \min(m-1, n-m).$$

Suppose that we wanted to compute the variance of $Z|y$ for all substrings y of x in accordance to the formula above. Applying the formula from scratch to each substring would require time $\Theta(|x|^3)$, since the number of possible distinct words appearing as substrings of x may be quadratic in $|x|$. In [15], the variance is computed for all prefixes of a string y in overall time $O(|y|)$, by making crucial use of a recurrence that speeds up computation of the term

$$B(m) = \sum_{l=1}^{s_m}(n-m+1-d_l)\Pi_{j=m-d_l+1}^{m} p_j.$$

In this expression, $B(m)$ refers to the prefix $y_1 y_2 ... y_m$ of some string y, $\mathcal{S}(m) = \{b_{l,m}\}_{l=1}^{s_m}$ is the set of borders "at m" associated with the periods of $y_1 y_2 ... y_m$ and $bord(m)$ is the longest border of $y_1 y_2 ... y_m$. By a simple adaptation of the `maxborder` it is possible to derive $B(m)$ quickly from knowledge of $bord(m)$ and of the previously computed values $B(1), B(2), ..., B(m-1)$. Specifically, letting the border associated with period d_l *at position* m to be

$$b_{l,m} = m - d_l,$$

the following expression of $B(m)$ holds:

$$B(m) = (n - 2m + 1 + bord(m))\Pi_{j=bord(m)+1}^{m} p_j$$

$$+2(bord(m) - m) \sum_{l=1}^{s_{bord(m)}} \Pi_{j=b_{l,bord(m)}+1}^{m} p_j$$

$$+ \ (\Pi_{j=bord(m)+1}^{m} p_j) \ B(bord(m)),$$

where the fact that $B(m) = 0$ for $bord(m) \leq 0$ yields the initial conditions. Note that each product of probabilities can be extracted in constant time from a precomputed table containing the products of the probabilities of all consecutive prefixes of x. From knowledge of $n, m, bord(m)$ and these prefix probability products, the first term of $B(m)$ is computed in constant time. Except for $(bord(m) - m)$, the second term is essentially a sum of probability products taken over all distinct borders of $y_1 y_2 ... y_m$. Thus, given such a sum and $B(bord(m))$ at this point enables one to compute $B(m)$ whence also the variance, in constant time. Maintaining knowledge of the value of such sums during the computation of longest borders is easy, since the value of the sum

$$T(m) = \sum_{l=1}^{s_{bord(m)}} \Pi_{j=b_{l,bord(m)}+1}^{m} p_j$$

obeys the recurrence:

$$T(m) = T(bord(m)) \cdot \Pi_{j=bord(m)+1}^{m} p_j + \Pi_{j=bord(bord(m))+1}^{m} p_j,$$

with $T(m) = 0$ for $bord(bord(m)) \leq 0$. In conclusion, the following holds.

Theorem 12 *Under the independently distributed source model, the mean and variances of all prefixes of a string can be computed in time and space linear in the length of that string.*

Application of this treatment to every suffix of a string yields the mean and variance of all substrings in overall optimal quadratic time. From what we have seen, the quest for surprising words under this model can be limited to those ending at the internal nodes of T_x. Since also the variances can be computed with our recurrence traveling backward on suffix links, this results in a global detector of unusual words in linear time and space.

5. FILTERING, FINGERPRINTING AND APPROXIMATE SEARCHING

The underlying theme of this section is the derivation of succinct albeit possibly approximate representations of objects. Hashing is one obvious way to do this. In an early approach to fast string searching, Karp, Miller and Rosenberg (cf. [67]) introduced a strategy based on some notion of a label or signature for the substrings of a string x, also called naming or numbering, as follows. First, generate the list of labels for individual characters, giving as a name to each character the position of its first occurrence in x. Next, perform approximately $\log|x|$ stages, as follows. At the k-th stage, compose all pairs of labels (l_i, l_{i+2^k}), sort them in lexicographic order and relabel each pair (whence also the substring it denotes) by the position of its first occurrence in the sorted list. If this process is performed on the concatenation of a pattern y and the text, then the occurrences of y can be intercepted subsequently by looking for positions of x with appropriate labels. We leave the details to the reader. Among its many virtues [67], this encoding has recently proved useful in capturing distant relationships among files for compression purposes [33].

Another notable approach to pattern searches based on hash signatures is due to Karp and Rabin [91]. The idea here is to first filter out candidates and then check them individually for exact matching. This philosophy represents a precursor for many strategies dealing with massive data.

In the filtering stage, the pattern y is hashed into a number and then a window of size $|y|$ is slided on the text while the hash values of the corresponding substrings are computed. To be effective in this context, the hash function must be highly discriminating for strings. At the same time, it should be quickly computed and updated in the transition from one text window to the next. This is met by assimilating the symbols of Σ with integers and defining the hash value h for string u by

$$h(u) = \left(\sum_{i=0}^{|u|-1} u[i] \times d^{|u|-1-i} \right) \bmod q,$$

where q and d are two constants. Then, for each string $v \in \Sigma^*$, and symbols $a', a'' \in \Sigma$, $h(va'')$ is computed from $h(a'v)$ by the formula

$$h(va'') = ((h(a'v) - a' \times d^{|v|}) \times d + a'') \bmod q.$$

During the search for pattern x, it suffices to compare the value $h(y)$ with the hash value associated with each substring of length m of text

p	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
$x[p]$	n	o	␣	d	e	f	e	n	s	e	␣	f	o	r	␣	s	e	n	s	e
$h(x[p...p+4])$	8	8	6	28	9	18	28	26	22	12	17	24	16	0	1	9	—	—	—	—

Figure 1.2 Illustrating Karp-Rabin's algorithm.

x. If these two values are equal, that is, in case of collision, it is still necessary to check whether the substring is equal to x or not by direct symbol comparisons.

Convenient values for d are the powers of 2; in this case, all products by d are computed as shifts on integers. The value of q is generally a large prime (such that the quantities $(q-1)\times d+|\Sigma|-1$ and $|\Sigma|\times q-1$ do not cause overflows), but it can also be the value of the implicit modulus supported by integer operations. The operation of the algorithm is illustrated in Figure 1.2, searching for the pattern $y = sense$ in the text $x =$ *no defense for sense*. Here, symbols are assimilated with their ASCII codes (hence $|\Sigma| = 256$), and the values of q and d are set respectively to 31 and 2. This is a valid choice when the maximal integer is $2^{16}-1$. The value of $h(y)$ is $(115\times 16+101\times 8+110\times 4+115\times 2+101) \bmod 31 = 9$. Since only $h(y[4...8])$ and $h(x[15...19])$ are equal to $h(y)$, only two substrings of x need to be checked. The worst-case complexity of this string-searching is quadratic, but a prudent choice of the values for q and p leads to $O(m+n)$ expected time.

Signatures may be used to obtain substrings that encapsulate a given text, but also strings that depart significantly from it. This is the general problem of *inverse pattern matching* [7], that refers to the task of inferring from a given textstring x a short pattern string y such that y is, by some measure, most typical (or, alternatively, most anomalous) in the context of x. This problem arises in a wide variety of applications and takes up numerous flavors, among which in particular those based on signatures or frequencies of pattern occurrences. When such occurrences need not be exact, alternative measures of typicality can be based on some notion of similarity among strings, such as the Hamming [87] or Levenshtein [98] distances. Given a textstring x and an integer m, for example, one might ask for a pattern y that scores the smallest (or largest) total number of mismatches when aligned with all substrings of x. Noteworthy variants of the problem arise when the constraint is added that y must be a substring of x, or, symmetrically, that y must not have any occurrence in x. Efficient (occasionally, optimal) sequential algorithms for the problem and its variants were provided in [7, 80]. Computations of these and similar "distance preserving signatures" (see *e.g.* [83]) find use in disparate contexts, including information retrieval, data compression, computer security and molecular biology. In the two latter fields, in particular, highly anomalous patterns are also often sought, *e.g.*, in intrusion [121] or plagiarism detection, in the synthesis of molecular probes in genome sequencing by hybridization [4], in designing control (inactive) antisense oligonucleotides, etc.

As an example, we illustrate the simplest (*min*) inverse pattern matching problem, which is defined as follows: given a text string $x = x_1 \cdots x_n$ and positive integer $m \leq n$, we want to produce a pattern string $y_{min} = y_1 \cdots y_m$ (of length m) where $ham(y_{min}, x) \leq ham(y, x)$ for all strings $y \in \Sigma^m$. The symmetric *(Max) Inverse Pattern Matching Problem* seeks instead a pattern y_{Max} such that $ham(y_{Max}, x) \geq ham(y, x)$ with respect to all $y \in \Sigma^m$. Both versions of the problem are solved by the same basic strategy. The naive algorithm for the min inverse pattern matching problem is computing the Hamming distance for every possible substring of length m, and choosing the minimum. This algorithm is clearly bad since it takes exponential time. However, an optimal algorithm for solving the problem is readily set up. The idea is to "synthesize" y by choosing its characters one at a time, in such a way that each character will maximize the matches when meeting the positions of the text it will face.

The most difficult variant of the problem is the Max external one, in which y is required not to appear in x. However, also this variant has been shown to have an optimal linear time solution [80].

5.1 APPROXIMATE SEARCHES

A natural departure from the problem of *exact* string searching, consists in assuming that a symbol can (perhaps only at some definite positions) match a small group of other symbols. At one extreme we may have, in addition to the symbols in the input alphabet Σ, a *don't care* symbol ϕ with the property that ϕ matches any other character in Σ. This gives rise to variants of string searching where, in principle, ϕ appears (i) only in the pattern, (ii) only in the text or (iii) both in pattern and text. Here we briefly address the main variant (i).

One approach to this variant is to try and extend one of the fast string searching algorithms by accommodating don't cares in the pattern. However, the obvious transitivity on character equality, that subtends those and other exact string searching, is lost with don't cares. Some partial recovery is possible when the number and positions of don't cares is fixed. In this case, one may think of adapting some multiple pattern automaton of the kind discussed earlier.

Manber and Baeza-Yates [104] considered the case where the pattern embeds a string of at most k don't cares, *i.e.*, has the form $y = u\phi^i v$, where $i \leq k$, $u, v \in \Sigma^*$ and $|u| \leq m$ for some given k, m. Their algorithm is off-line in the sense that the text x is preprocessed to build the suffix array associated with it. This operation costs $O(n \log n)$ time in the worst case. Once this is done, the problem reduces to one of efficient implementation of 2-dimensional orthogonal range queries.

A landmark paper by Fischer and Paterson [76] exposed the similarity of string searching to multiplication, thereby obtaining a number of interesting algorithms for exact string searching and some of its variants. It is not difficult to see that string matching problems can be rendered as special cases of a general linear product. Given two vectors X and Y, their *linear product* with respect to two suitable operations $\otimes$ and $\oplus$, is denoted by $X \overset{\otimes}{\underset{\oplus}{}} Y$, and is a vector $Z = Z_0 Z_1 \ldots Z_{m+n}$ where $Z_k = \bigoplus_{i+j=k} X_i \bigotimes Y_j$ for $k = 0, \ldots, m+n$. If we interpret $\oplus$ as the boolean $\wedge$ and $\otimes$ as the symbol equivalence $\equiv$, then a match of the reverse Y^R of Y, occurs ending at position k in X, where $m \leq k \leq n$, if $[X_{k-m} \ldots X_k] \equiv [Y_m \ldots Y_0]$, that is, with obvious meaning, if $(X \overset{\equiv}{\underset{\wedge}{}} Y)_k$ =TRUE. This observation brings string searching into the family of boolean, polynomial and integer multiplications thereby leading quickly to an $O(n \log m \log \log m)$ time solution even in the presence of don't cares, provided that the size of Σ is fixed.

Some central notions of similarity are based on three basic *edit* operations on strings. Given any string w we consider the *deletion* of a symbol from w, the *insertion* of a new symbol in w and the *substitution* of one of the symbols of w with another symbol from Σ. It may be assumed that each edit operation has an associated nonnegative real number representing the *cost* of that operation, so that the cost of deleting from w an occurrence of symbol a is denoted by $D(a)$, the cost of inserting some symbol a between any two consecutive positions of w is denoted by $I(a)$ and the cost of substituting some occurrence of a in w with an occurrence of b is denoted by $S(a, b)$.

Letting now x and y be two strings of respective lengths $|x| = n$ and $|y| = m \leq n$, the *string editing problem* for input strings x and y consists in finding a sequence of edit operations or *edit script* Γ of minimum cost that transforms y into x. The cost of Γ is the *edit distance from y to x*. Edit distances where individual operations are assigned integer or unit costs occupy a special place. Such distances are often called Levenshtein distances, since they were introduced by W. Levenshtein in connection with error correcting codes [98]. String editing finds applications in a broad variety of contexts, ranging from speech processing to geology, from text processing to molecular biology.

It is not difficult to see that the general (*i.e.*, with unbounded alphabet and unrestricted costs) problem of edit distance computation is solved by a serial algorithm in $\Theta(mn)$ time and space, through dynamic programming. Due to its widespread application of the problem, however, such a solution and a few basic variants were discovered and published in a diverse literature (cf., *e.g.* [21]). An $\Omega(mn)$ lower bound was established for the problem by Wong and Chandra for the case where the queries on symbols of the string are restricted to tests of equality. For unrestricted tests, a lower bound $\Omega(n \log n)$ was given by Hirschberg. Algorithms slightly faster than $\Theta(mn)$ were devised by Masek and Paterson, through resort to the so-called "Four Russians Trick". The "Four Russians" are Arlazarov, Dinic, Kronrod, and Faradzev. Along these lines, the total execution time becomes $\Theta(n^2/\log n)$ for bounded alphabets and $O(n^2(\log\log n)/\log n)$ for unbounded alphabets. The method applies only to the classical Levenshtein distance metric, and does not extend to general cost matrices. To this date, the problem of finding either tighter lower bounds or faster algorithms is still open. Details and references can be found in, *e.g.*, [20, 21]).

The computation of edit distances by dynamic programming is readily set up. For this, let $C(i,j)$, $(0 \le i \le |y|,\ 0 \le j \le |x|)$ be the minimum cost of transforming the prefix of y of length i into the prefix of x of length j. Let w_k denote the kth symbol of string w. Then $C(0,0) = 0$, $C(i,0) = C(i-1,0) + D(y_i)$ $(i = 1,2,...,|y|)$, $C(0,j) = C(0,j-1) + I(x_j)$ $(j = 1,2,...,|x|)$, and $C(i,j)$ will be given by

$$\min\{C(i-1,j-1) + S(y_i,x_j),\ C(i-1,j) + D(y_i),\ C(i,j-1) + I(x_j)\}$$

for all i,j, $(1 \le i \le |y|, 1 \le j \le |x|)$. Observe that, of all entries of the C-matrix, only the three entries $C(i-1,j-1)$, $C(i-1,j)$, and $C(i,j-1)$ are involved in the computation of the final value of $C(i,j)$. Hence $C(i,j)$ can be evaluated row-by-row or column-by-column in $\Theta(|y||x|) = \Theta(mn)$ time. An optimal edit script can be retrieved at the end by backtracking through the local decisions that were made by the algorithm.

A few important problems are special cases of string editing, including the *longest common subsequence* problem, *local alignment* , *i.e.*, the detection of local similarities of the kind sought typically in the analysis of molecular sequences such as DNA and proteins, and some important variants of *string searching with errors*, or searching for approximate occurrences of a pattern string in a text string. As highlighted in the following brief discussion, a solution to the general string editing problem implies typically similar bounds for all these special cases.

In many cases of great practical interest, such as *e.g.*, with genomic sequence analysis, the space occupied by the edit distance matrix is unbearable and linear space methods are sought. We refer to [13, 21] for details and references.

Sequence similarity is a natural and useful filter for extracting matching information from huge data repositories. some of the fastest and most efficient searches routines work by first detecting regions of strong local resemblance, using conceptual tools of the kind represented by the following lemma.

Lemma 15 If x and y match with at most k differences, then x and y must have at least one identical substring of length $r = \lfloor max\{|x|,|y|\}/(k+1)\rfloor$

Proof. Let w.l.o.g. $|x| = max\{|x|,|y|\}$, and divide x into consecutive intervals of length r. In the alignment, each interval aligns to some part of y, determining $k+1$ subalignments. If each of these subalignments contained at least one error, then we would have more than k errors. Thus, at least one of the intervals must match exactly a corresponding interval of y. ◇

More about searching with errors is said in the next subsection.

5.2 STRING SEARCHING WITH ERRORS

Consider the problem of computing, for every position of the textstring x, the best edit distance achievable between y and a substring w of x ending at that position. Under the unit cost criterion, a solution is readily derived from the recurrence for string editing. The first obvious change consists in setting all costs to 1 except that $S(y_i, x_j) = 0$ for $y_i = x_j$. Thus, we have now, for all i, j, $(1 \leq i \leq |y|, 1 \leq j \leq |x|)$,

$$C(i,j) = \min\{C(i-1,j-1)+1,\ C(i-1,j)+1,\ C(i,j-1)+1\}.$$

A second change affects the initial conditions, so that we have now $C(0,0) = 0$, $C(i,0) = i$ $(i = 1,2,...,m)$, $C(0,j) = 0$ $(j = 1,2,...,n)$. This has the effect of setting to zero the cost of prefixing y by any prefix of x. In other words, any prefix of the text can be skipped free of charge in an optimum edit script.

The computation of C is then performed in much the same way as before, thus taking $\Theta(|y||x|) = \Theta(mn)$ time. This time around, we are interested in the entire last row of matrix C at the outset.

In practice, it is often more interesting to locate only those segments of x that present a high similarity with y under the adopted measure. Formally, given a pattern y, a text x and an integer k, this restricted version of the problem consists in locating all terminal positions of substrings w of x such that the edit distance between w and y is at most k.

The recurrence given above will clearly produce this information. However, there are more efficient methods to deal with this restricted case. In fact, a time complexity $O(kn)$ and even sublinear expected time are achievable. We refer to, *e.g.*, [20, 65] for detailed discussions. In the following, we review some basic principles subtending an $O(kn)$ algorithm for string searching with k differences. Note that when k is a constant the corresponding time complexity is linear.

The crux of the method is to limit computation to $O(k)$ elements in each diagonal of the matrix C. These entries will be called *extremal* and may be defined as follows: a diagonal entry is d-extremal if it is the deepest entry on that diagonal to be given value d ($d = 1, 2, ..., k$). Note that a diagonal might not feature any, say, 1-extremal entry, in which case it would correspond to a perfect match of the pattern. The identification of d-extremal entries proceeds from extension of entries already known to be $(d-1)$-extremal. Specifically, assume we knew that entry $C(i, j)$ is $(d-1)$-extremal. Then, any entry reachable from $C(i, j)$ through a unit vertical, horizontal or diagonal-mismatch step possibly followed by a maximal diagonal stream of matches is d-extremal at worst. In fact, the cost of a diagonal stream of matches is 0, whence the cost of an entry of the type considered cannot exceed d. On the other hand, that cost cannot be smaller than $d - 1$, otherwise this would contradict the assumption $C(i, j) = d-1$. Let entries reachable from a $(d-1)$-extremal entry $C(i, j)$ through a unit vertical, horizontal or diagonal-mismatch step be called *d-adjacent*. Then the following program encapsulates the basic computations.

Algorithm `k-err` **:**
element array $x[1 : n], y[1 : m], C[0 : m; 0 : n]$; integer k
begin
($PHASE$ 1 : initializations)
  set first row of C to 0;
  find the boundary set S_0 of 0-extremal entries
  by exact string searching;
($PHASE$ 2 : identify k-extremal entries)
  for d = 1 **to** k **do**
    begin
    walk one step horizontally, vertically and
    (on mismatch) diagonally
    from each $(d-1)$-extremal entry in set $S_{(d-1)}$
    to find d-adjacent entries;
    from each d-adjacent entry, compute the farthest
    d-valued entry reachable diagonally from it;
    end

for $i\ =\ 1$ **to** $n-m+1$ **do**
  begin
  select lowest d-entry on diagonal i
  and put it into the set S_d *of d-extremal entries*
  end
end.

It is easy to check that the algorithm performs k iterations in each one of which it does essentially a constant number of manipulations on each of the n diagonals. In turn, each one of these manipulations takes constant time except at the point where we ask to reach the farthest d-valued entry from some other entry on a same diagonal. We would know how to answer quickly that question if we knew how to handle the following query: given two arbitrary positions i and j in the two strings y and x, respectively, find the longest common prefix between the suffix of y that starts at position i and the suffix of x that starts at position j. In particular, our bound would follow if we knew how to process each query in constant time. It is not known how that could be done without preprocessing becoming somewhat heavy. On the other hand, it is possible to have it such that *all* queries have a cumulative amortized cost of $O(kn)$. This possibility rests on efficient algorithms for performing *lowest common ancestor* queries in trees. Space limitations do not allow us to belabor this point any further.

In massive applications, even time $O(kn)$ may be prohibitive. Using filtration methods it is possible to set up sublinear expected time queries. As already highlighted, one possibility is to first look for regions with exact replicas of some pattern segment and then scrutinize those regions. Another, is to look for segments of the text that are within a small distance of some fixed segments of the pattern. Some of the current top performers in molecular database searches are engineered around these ideas [5, 131, 26, 53]. In fact, the whole issue of filtration search may be regarded as a form of pattern discovery [42, 41, 39, 40], probably a fundamental application of future Pattern Matching and one that is discussed more extensively later in this chapter.

The special case where insertions and deletions are forbidden is also solved by an algorithm very similar to the above and within the same time bound. This variant of the problem is often called string searching *with mismatches.* A probabilistic approach to this problem is implicit in [53]. When k cannot be considered a constant, an interesting alternative results from Abrahamson's approach to multiple-value string searching [1] which results in an algorithm of time $O(nm^{1/2} \log m \log \log^{1/2} m)$.

6. COMPRESSING, LEARNING, MINING, AND DISCOVERING

Data compression brings savings in storage space and transmission time, two commodities in increasingly scarce supply. From the perspective of the data flood ahead, compression also helps in the formation of succinct descriptors and models, thereby helping in overcoming the ultimate limitations imposed by the narrow bandwidth of the final user. Because of this, efficient, innovative compression methods will continue to play an important role.

Of the two main broad classes of compression, standard *lossy* methods such as Mpeg, Jpeg, Wavelets etc. have a definite numerical flavor and derive a limited influence from Pattern Matching. By contrast, nearly every present and future *lossless* method will use more or less sophisticated Pattern Matching techniques. Among the basic methods in this class, we find Run-Length and Huffman Encoding, the latter being further subdivided into static and dynamic codes, Arithmetic Codes, Macro Schemes such as the Ziv-Lempel methods underlying *compress, gzip* and other popular tools, the more recent Burrows-Wheeler transform subtending *bzip*, Predictive Codes, etc. These and others are reviewed in this section.

6.1 STANDARD COMPRESSION METHODS

We outline here some classical yet practical text compression algorithms. Algorithmic efficiency is but one of the parameters against which the efficiency of a method is assessed. The final compression ratio is equally, if not more, important. This latter depends on the nature of the input data. Typically, the final size of compressed textfiles vary from 30% to 50% of the size of the input.

In standard lossless compression, two main strategies are applied. The first strategy is a statistical method that takes into account the frequencies of symbols to build a uniquely decipherable code optimal with respect to the compression. This is considered in Subsection 6.1.1. Subsection 6.1.2 presents a refinement of the coding algorithm of Huffman based on the binary representation of numbers. Huffman codes contain new codewords for the symbols occurring in the text. In this method, fixed-length blocks of bits are encoded by different codewords. In the second strategy, repeated substrings of variable-length from the text are spotted and suitably encoded. This will be seen in Subsection 6.1.3.

Due to its ability to capture context dependency, this second strategy often provides better compression ratios.

6.1.1 Huffman coding. The Huffman method is an optimal statistical coding, in which each character or fixed block of characters of the text is replaced by a *codeword* in such a way, that longer and longer codewords are assigned to rarer and rarer characters. The method works for any block length, however, the running time grows exponentially with the length.

The Huffman algorithm uses *prefix* codes, *i.e.*, sets of words in which no word is a *prefix* of another. The advantage with such codes is that decoding is *instantaneous*, in the sense that it can be carried out while the encoded string is being received.

A prefix code on the alphabet $\{0, 1\}$ is represented in a natural way by a binary digital trie in which the leaves are labeled by the original characters, and the path from the root to a character spells out the characters codeword. The specific assignment of codewords depends on the frequencies of the individual characters. The complete compression algorithm consists of three stages: count of character frequencies, construction of the prefix code, encoding of the text. The last two steps use information computed by their preceding step. Decoding is a simple exercise.

The static Huffman method has two main drawbacks: first, if the frequencies of characters in the source text are not known *a priori*, then the input text has to be read twice; second, the coding tree must be included in the compressed file. This is avoided by dynamically updating the coding tree for the consecutive prefixes of the text while consecutive symbols are processed. A combinatorial property of the tree allows its efficient updates. By mimicking the coding process, decoding will expose the tree precisely in the same order.

6.1.2 Arithmetic coding. In arithmetic coding, symbols are treated as digits of a numeration system, and texts as decimal parts of numbers between 0 and 1. The interval $[0, 1[$ is first partitioned into $|\Sigma|$ subintervals of size proportional to the probabilities or frequencies of symbols. The same partition is then recursively applied to subintervals as consecutive text symbols are read, thereby mapping the text itself into some subinterval of $[0, 1[$. Compression is achieved because highly probable texts end up mapped in wider intervals thus requiring fewer bits in their description.

Formally, let the interval associated with symbol $a_i \in \Sigma$ $(1 \leq i \leq \|\Sigma|)$ be denoted $I(a_i) = [l_i, h_i[$. The intervals satisfy the conditions: $l_1 = 0$,

$h_{|\Sigma|} = 1$, and $l_i = h_{i-1}$ for $1 < i \leq |\Sigma|$. Note that $I(a_i) \cap I(a_j) = \emptyset$ if $a_i \neq a_j$.

The encoding consists in computing the interval corresponding to the input text. We begin with the initial interval $[0, 1[$. The generic step deals with a symbol a_i of the source text by transforming the current interval $[l, h[$ into $[l', h'[$ where $l' = l + (h - l) * l_i$ and $h' = l + (h - l) * h_i$. From a theoretical standpoint, l alone would suffice to encode the input text.

The decoding phase recapitulates the encoding. Specifically, the first step of decoding consists in identifying the symbol a_i such that $l \in I(a_i)$. At that point, l is replaced by

$$l' \leftarrow \frac{l - l_i}{h_i - l_i},$$

and the process is repeated until $l = 0$. The main problem with arithmetic coding is coping with finite precision while performing arithmetics on real numbers.

6.1.3 LZW Coding. Ziv and Lempel designed a class of compression methods based on the idea of self reference: while the textfile is scanned, substrings or *phrases* are identified and stored in a dictionary, and whenever, later in the process, a phrase or concatenation of phrases is encountered again, this is compactly encoded by suitable pointers [97, 138, 139]. Of the several existing versions of the method, we describe below the one known as Lempel-Ziv-Welsh method, which is incarnated by the `compress` feature under the UNIX operating system.

For the encoding, a dictionary is initialized with all the characters of the alphabet. At the generic iteration, we have just read a segment w of the text. With a the symbol following this occurrence of w, we now proceed as follows: If wa is in the dictionary we read the next symbol, and repeat with segment wa instead of w. If, on the other hand, wa is not in the dictionary, then we append the dictionary index of w to the output file, and add wa to the dictionary; then reset w to a and resume processing from the text symbol following a. Once w is initialized to be the first symbol of the source text, "w belongs to the dictionary" is established as an invariant in the above loop.

Decoding is symmetric, in particular, the dictionary is recovered while the decompression process runs. The basic routine is as follows. We start with a basic dictionary of symbols. Then, when we read the encoding c from the compressed file, we write to the output file the segment w having index c in the dictionary, and add to the dictionary the word wa

where a is the first letter of the next segment. Except for a special case, note that we can infer the appropriate dictionary index for wa. A very special case requiring extra care occurs if the symbol a is also the first symbol of w. It is indeed related to squares that occur in the text. We leave the analysis of this case and its (easy) recovery for an exercise.

6.1.4 The Burrows-Wheeler Transform. A recent, imaginative approach due to M. Burrows and D.J. Wheeler [50] successfully exploits the delicate interplay between locality of reference and pointer size. Assuming an input string $x = dadcbbe$, the encoding performs the following steps. First, we build a table of the cyclic shifts of x, as follows.

S0	*d a d c b b e*
S1	*a d c b b e d*
S2	*d c b b e d a*
S3	*c b b e d a d*
S4	*b b e d a d c*
S5	*b e d a d c b*
S6	*e d a d c b b*

Next, these rotations are lexicographically sorted, resulting in the table:

S1	*a d c b b e d*
S4	*b b e d a d c*
S5	*b e d a d c b*
S3	*c b b e d a d*
S0	*d a d c b b e*
S2	*d c b b e d a*
S6	*e d a d c b b*

It turns out that strings like the string $y = dcbdeab$ in the last column are highly compressible, *e.g.*, by run-length. In fact, the first column contains sorted symbols that are each immediately adjacent in x to the corresponding symbol in the last column. It is expected then that, in correspondence with a run on the first column, the last one also contains a run. Note that it is possible to go back from the last column y to the first column $y' = abbcdde$ simply by sorting y. More importantly, from knowledge of y, y' and of the rank i of the original string in the sorted list, it is possible to reconstruct the original sequence x. This is achieved by setting up a suitable transformation vector T that tells, for each row j, where in x is row $j + 1$. This vector can be figured out by looking at y and y' as shown in the table below.

0 S1	*d*	*a*
1 S4	*c*	*b*
2 S5	*b*	*b*

3 S3	*d*	*c*
4 S0	*e*	*d*
5 S2	*a*	*d*
6 S6	*b*	*e*

Clearly, we have $T(4) = 0$ since c moves, but what about row 1? The b there could go to either row 0 or 1. The important property is, since y' is sorted then rows beginning with a same character are also sorted. Thus, the first b in row 1 moves to row 0, the second b comes from row 6. The final touch of the method is to perform move-to-front encoding of y. In practice, all 256 codes are kept in a list, and each time a character is to be output, its position is sent to the list, then moved to the front. The result is a string with many of 0's and small integers,which can be compressed using entropy encoders. For example, $y = tttWtwttt$ would be encoded as $[116, 0, 0, 88, 1, 119, 1, 0, 0]$.

The sorting inherent to the Burrows-Wheeler method is suitably implemented with suffix arrays, resulting in a relatively fast process.

6.2 DATA COMPRESSION USING ANTIDICTIONARIES

Yet another basic text compression method, called *DCA*, uses some "negative" information about the text, which is described in terms of antidictionaries [62, 63, 64]. Contrary to the Ziv and Lempel methods that are centered on dictionaries or sets of words occurring as substrings in the text, this method takes advantage from words that *do not* occur as substrings in the text and are said to be *forbidden*. It is natural to call such sets of words *antidictionaries*.

6.2.1 Encoding and decoding. Let x be the text on a binary alphabet and let $F(x)$ be the set of substrings of x. For instance, if $x = \texttt{01001010}$ then $F(x) = \{\varepsilon, \texttt{0}, \texttt{1}, \texttt{00}, \texttt{01}, \texttt{10}, \texttt{001}, \texttt{010}, \texttt{100}, \texttt{101}, \ldots\}$. The antidictionary AD is a *factor code* (no word of the set is a substring of another word of the set) included in $\Sigma^* \setminus F(x)$. For example, $\{\texttt{000}, \texttt{10101}, \texttt{11}\}$ is an antidictionary for $x = \texttt{01001010}$.

The compression algorithm processes the input file on-line. At the generic step, we have read some prefix w of x, and inspect the symbol, say, a, that immediately follows w. If there exists a word $u \in AD$ that is a suffix of wa, then the symbol a is deleted, since it is predictable through resort to the antidictionary. The compression algorithm based on this principle is listed below. In order to be able to decode the output of the encoder, an additional mechanism is necessary. To simplify the exposition, we assume here that the encoder produces also the length of the original text. The decoder works in a fashion which is dual to the

ENCODER (anti-dictionary AD, word $x \in \{\mathtt{0}, \mathtt{1}\}^*$)
1. $\gamma \leftarrow \varepsilon$;
2. **for** $a \leftarrow$ first to last symbol of x
3.   **if** for any suffix v of the processed text, $v\mathtt{0}, v\mathtt{1} \notin AD$
4.   output a;
5. **return** $(|x|, \gamma)$;

DECODER (anti-dictionary AD, integer n, word $\gamma \in \{\mathtt{0}, \mathtt{1}\}^*$)
1. $w \leftarrow \varepsilon$;
2. **while** $|w| < n$
3.   **if** for some suffix v of w and some $a \in \{\mathtt{0}, \mathtt{1}\}$, $va \in AD$
4. $w \leftarrow w \cdot \neg a$;
5.   **else**
6. $b \leftarrow$ next symbol of γ;
7. $w \leftarrow w \cdot b$;
8. **return** (w);

Figure 1.3 Antidictionary based compression

encoder, and is presented immediately following it. It uses its knowledge of the length in order to decide when to halt.

The advantage of having a factor code as antidictionary is that the test at Line 3 in the decoder can be satisfied by only one word va. Therefore, no useless word is stored in the antidictionary.

6.2.2 Implementing finite antidictionaries. The antidictionary queries invoked by the above algorithms are implemented as follows. Starting with the trie of words in the antidictionary, the automaton $\mathcal{A}(AD)$ is built that accepts all strings of which no substring appears in the antidictionary. This is an application of the Aho-Corasick algorithm to the trie, and results in a linear-time algorithm. With this automaton in place, and while reading the text to encode, whenever a transition leads to a state associated with a word of the antidictionary the decoder outputs the dual symbol.

The automaton $\mathcal{A}(AD)$ can be easily transformed into a (finite-state) transducer $\mathcal{T}(AD)$ that realizes the compression algorithm. The decompression may be similarly realized by a dual transducer, which is obtained by interchanging input and output labels in the first transducer (with an additional halting instruction to stop the decoding).

The automaton $\mathcal{A}(AD)$ (or the transducer $\mathcal{T}(AD)$) has an interesting synchronization property, which makes it possible to develop algorithms to search compressed texts or to design parallel version of the encoding and decoding algorithms. With k the maximal length of words in AD, this property is as follows: given any two paths $(q_1, a_1, q_2) \cdots (q_k, a_k, q_{k+1})$ and $(q'_1, a_1, q'_2) \cdots (q'_k, a_k, q'_{k+1})$ having the same label $a_1 \cdots a_k$, then the two ending states q_{k+1} and q'_{k+1} coincide. Thus, the encoding of a part of the text certainly depends on its left context, but this is limited to up to a length of k only.

6.2.3 How to build Antidictionaries. In practical applications, the antidictionary is not given *a priori* but it must be derived either from the text to be compressed or from a family of texts produced by the same source as the one producing the text. There exist several criteria to build efficient antidictionaries, that variously depend on different aspects or parameters that one wishes to optimize in the compression process. In turn, each criterion gives rise to a different algorithm and implementation.

The general methods to build antidictionaries are based on data structures that store substrings of words, such as suffix tries, suffix trees, dawgs, and suffix or factor automata. As seen, some notion of a suffix link always accompanies these structures, and such a notion proves essential also in designing efficient algorithms to build representations of sets of minimal forbidden words in term of tries or trees. The antidictionary constructions set up along these lines take time linear in the length of the text to be compressed.

A rough solution to control the size of antidictionaries would be obviously by bounding the length of the words that are admitted in it. A better solution in the static compression scheme is to prune the trie of the antidictionary on the basis of a tradeoff between the space of the trie to be transmitted and the gain in compression. However, the first solution is enough to get compression rates that reach asymptotically the entropy for balanced sources, even if this is not true for general sources. Both solutions can be engineered to run in linear time.

A further sophistication considers self-compressed antidictionaries, and yields best compression ratios for the method.

6.2.4 Variations. The static compression scheme presented above requires to read the text twice. Several variations and improvements can be elaborated upon based on clever combinations of two features suitably injected in the model, namely, statistical filters and dynamic implementations. These are classical features, often included in most data compression methods.

Statistical considerations can be used in the construction of antidictionaries. If a forbidden word is responsible for erasing few bits of the text in the compression algorithm while its description as an element of the antidictionary is "expensive", then the compression rate improves by excluding that word from the antidictionary. On the other hand, one can introduce in the antidictionary a word that is not forbidden but occurs very rarely in the text. In this case, the compression algorithm may produce some errors in predicting the next letter. In order to keep a lossless compression scheme, encoder and decoder must be adapted to manage such errors. Typical errors occur in the case of antidictionaries built for fixed sources as well as in the dynamic approach. Even with errors, assuming that they are rare with respect to the longest word (length) of the antidictionary, the compression scheme may be shown to preserve the synchronization property.

6.3 SEARCHING COMPRESSED TEXT

For data stored in compressed form, navigation through compressed databases poses additional questions of string pattern matching. The first question is whether it may be more efficient to decompress the data before processing a search or other standard query or, given the possibility, it might be more expedient to perform the query directly on the compressed data. The answer depends of course on the particular problem instance, as well as on such variables as compression method, algorithmic complexity, memory space available, etc. Among the various methods, the Ziv-Lempel family of compressors have received the largest attention, beginning with studies by Amir, Benson and Farach [8] and Farach and Thorup [71]. Along these lines, string search in compressed text was developed for the paradigm by Ziv and Lempel [138] and its subsequent variant by Welch [136]. The complexities for the searches are respectively of $O(n \log n' + m)$ and $O(n \log m + m)$, where n' is the size of the decompressed text and m the size of the pattern. Thus, compared to linear time string searching in plain texts, an extra log factor emerges. For large patterns, it makes sense to consider instances of the problem where also the pattern is compressed. This case was studied by Gąsieniec and Rytter [81], who gave algorithms respectively of time $O((n+m)^5)$ and $O((n+m)\log^c(n+m))$ (with c a positive constant) for the LZ and LZW compressors.

Searching files compressed by Huffman encoding is a classical problem treated, *e.g.*, in [111]. Shibata *et al.* [125] give a linear-time searching algorithm for files compressed by using antidictionaries.

Mixed techniques have also been developed in which the compression is designed to reduce the searching time. Examples of this approach may be found in [103] and [113]. The main drawbacks with the technique is that it often leads to less efficient compression and that of course it will not work with text compressed by standard methods.

6.4 LEARNING PROBABILISTIC AUTOMATA AND MODELING BY MARKOV CHAINS

Compression is but one of the domains within which the need arises to develop models of sources. In fact, as already mentioned, the statistical modeling of sequences is a central paradigm of machine learning that finds multiple uses in many domains. The probabilistic automata typically built in these contexts are subtended by uniform, fixed-memory Markov models. In practice, such automata tend to be bulky and computationally imposing both during their synthesis and use. In [120], much more compact, tree-shaped variants of probabilistic automata are described which assume an underlying Markov process of variable memory length. These variants, called *PST*s were successfully applied to learning and prediction of protein families in [29].

In one such automaton, each edge is labeled by a symbol, each node corresponds to a unique string —the one obtained by traveling from that node to the root— and nodes are weighted by a probability vector giving the distribution over the next symbol. The construction starts with a tree consisting of just the root node (*i.e.*, the tree associated with the empty word) and adds paths as follows. It considers the substrings from a family S of strings in order of increasing length. For each substring s considered, it is checked whether there is some symbol σ in the alphabet for which the empirical probability of observing it in S after s is significant and significantly different from the probability of observing it after the longest suffix $suf(s)$ of s. Whenever these conditions hold, the path relative to the substring (and possibly its necessary but currently missing ancestors) is added to the tree.

Given now a string, its weighting by a tree is done by scanning the string one letter after the other while assigning a probability to every symbol, in succession. The probability of a symbol is calculated by walking down the tree in search for the longest suffix that appears in the tree and ends immediately before that symbol, and multiplying the corresponding conditional probability. Since, following each input symbol, the search for the deepest node must be resumed from the root, this process cannot be carried out on-line nor in linear-time in the length of the tested sequence.

As is easy to see, the process of learning the automaton from a given training set S of sequences requires $\Theta(Ln^2)$ worst-case time, where n is the total length of the sequences in S and L is the length of a longest substring of S to be considered for a candidate state in the automaton. Once the automaton is built, predicting the likelihood of a query sequence of m characters may cost time $\Theta(m^2)$ in the worst case. A more efficient computation of empirical probabilities and conditional probabilities, of the kind described in an earlier section of this chapter, leads to equivalent automata that can be learned in time linear in the input size, and will subsequently predict a string of m symbols in $O(m)$ time. We refer to [14] for details.

6.5 EPISODES AND AUTOMATIC ASSOCIATION GENERATION

Many interesting problems can be cast in the emerging contexts of *data mining* and *information extraction*. As is well known, while traditional data base queries aim at retrieving records based on their isolated contents, these contexts focus on the identification of patterns occurring across records, and aim at the retrieval of information based on the discovery of interesting rules present in large collection of data. Central to these developments is the notion of an *association rule*, which is an expression of the form $S_1 \to S_2$ where S_1 and S_2 are sets of data attributes endowed with sufficient *confidence* and *support*. Sufficient support for a rule is achieved if the number of records whose attributes include $S_1 \cup S_2$ is at least equal to some pre-set minimum value. Confidence is measured instead in terms of the ratio of records having $S_1 \cup S_2$ over those having S_1, and is considered sufficient if this ratio meets or exceeds some pre-set minimum. Clearly, a statistic of the number of records endowed with the given attributes must be computed as a preliminary step, and this is often a bottleneck for the process of information extraction. We refer to [2] and [115] for a broader discussion of these concepts.

Some of the considerations developed earlier in this chapter may be regarded from a perspective of automatic generation of association rule. Lemma 12, for instance, can be rephrased by saying that for every word ending in the middle of an arc in T_x, a rule is exposed whereby any occurrence of that word in x *implies* an occurrence also of its extension to the nearest node. From this perspective, the construction of the tree may be regarded as a means for the discovery of this rule.

In a real discovery, though, we do not know *a priori* the rule that will be discovered. Along these lines, looking for squares, palindromes, *etc.* is only half a discovery, in so far as the "rule" (*e.g.*, ww, ww^R) which we are after is known beforehand. Even so, some mild extensions of this problem may already fit the mining paradigms.

For example, consider the problem of finding, for a given textstring x of n symbols and an integer constant d, and for any pair (y, z) of subwords of x, the number of times that y and z occur in tandem (*i.e.*, with no intermediate occurrence of either one in between) within a distance of d positions of x. Although in principle there might be n^4 distinct subword pairs in x, Lemma 12 tells us that it suffices to consider a family of only n^2 such pairs, with the property that for any neglected pair (w', z'), there is a corresponding pair (y, z) contained in our family and such that: (i) w' is a prefix of w and z' is a prefix of z, and (ii) the tandem index of (w', z') equals that of (w, z). We leave it as an exercise for the reader to find an efficient algorithm for the construction of the table of all such tandem indices. The particularization of the problem to the tandem index of occurrences of the same pattern, which is in fact a relaxed square detection problem, has also been studied recently [49].

A. Amir et al. [10] have used tries to organize and speed up the discovery of association rules in a typical data base, the entries of which are sets of attributes. The first step consists in transforming each record into a string by numbering the different attributes. Next, every set is considered as a string sorted by order of the attribute number. At this point, a trie is built by incremental insertion of all i-elements sorted sets for $i = 1, 2, ... i_{max}$, in succession, where i_{max} is some suitable bound. The nodes of the trie are weighted by the count of the number of records leading to each node (a measure of the support for that node). The data structure at the outset encodes all potential *covers*, a cover in this context being a set of attributes with support exceeding a certain *minsupport* value. To generate associations, one observes that once an association of the form $S \rightarrow \{a\}$ is generated for an attribute, this gives a handle to narrowing down the space of potential attributes of the form $\{a, b\}$, in the sense that only if both associations $S \cup \{a\} \rightarrow \{a\}$ and $S \cup \{b\} \rightarrow \{b\}$ exist, one can hope for association $S \rightarrow \{a, b\}$ to exist. This leads to the following scheme for association generations.

- For each node of the trie, let $s = s_1 s_2 ... s_k$ be the label of the path from the root to that node. Extract, in succession, each s_i and check the resulting string $\bar{s}$ for its support. Whenever the ratio $supp(s)/supp(s_1 ... s_{i-1} s_{i+1} ... s_k) \geq minconf$ then $S - S_i \rightarrow S_i$ is an *association rule.*

- We now have association rules with only one set on the right hand side. These rules are now combined to generate multiple rules. That is, for every pair of rules, generate a new rule with a consequent of size 2, and test its confidence level. Repeat the process to obtain rules with consequents of increasing size.

Other discoveries can be modeled in terms of the detection of special kinds of subsequences. A pattern $y = y_1 \dots y_m$ occurs as a *subsequence* of a text $x = x_1 \dots x_n$ iff there exist indices $1 \leq i_1 < i_2 < \cdots < i_m \leq n$ such that $x_{i_1} = y_1$, $x_{i_2} = y_2$, $\cdots$, $x_{i_m} = y_m$; in this case we also say that the substring $w = x_{i_1} x_{i_1+1} \dots x_{i_m}$ of x is a *realization* of y beginning at position i_1 and ending at position i_m in x. Given two strings $x = x_1 \dots x_n$ and $y = y_1 \dots y_m$ over an alphabet Σ, the problem of testing whether y occurs as a subsequence of x is trivially solved in linear time. It is also known that a simple $O(n \log |\Sigma|)$ time preprocessing of x makes it easy to decide subsequently for any x and in at most $|y| \log |\Sigma|$ character comparisons, whether y is a subsequence of x. These problems become more complicated if one asks instead whether y occurs as a subsequence of some substring w of x of bounded length. One way to answer the question is by identifying all distinct minimal realizations w of y. By a realization w being minimal with respect to x, it is meant that y is not a subsequence of any proper substring of w. Variants of this problem arise in numerous applications, ranging from information retrieval and mining recurrent events in telecommunications (see, *e.g.*, [106]) to molecular sequence analysis (see, *e.g.*, [134]) and intrusion and misuse detection in a computer system. Algorithms for the so-called *episode matching* [106] problem, which consists in finding the *earliest* occurrences of y in all minimal realizations w of y in x have been given in [68]. An occurrence $i_1 i_2 \dots i_m$ of y in a realization w is an earliest occurrence if the string $i_1 i_2 \dots i_m$ is lexicographically smallest with respect to any other possible occurrence of y in x. The algorithms in [68] perform within roughly $O(nm)$ time, without resorting to any auxiliary structure or index based on the structure of the text.

Many modern pattern or *motif* characterizations and discovery algorithms will come from the flourishing area of Bioinformatics, a microcosmos within which most problems of managing the data and information flood find early and somewhat controlled reflections (see, *e.g.*, [42, 41, 39, 40]). Prominent in this context is the issue of aligning multiple sequences [21]. This application is explosive in computational demand and is typically approached by way of heuristics. These, in turn, are variously centered around ideas of hinging putative alignments around similar subpatterns of various kinds. One difficulty in this regard is the lack of a unified notion of global comparison, which compounds with the inherent intractability of most exact methods. One way to approach the problem is then to look for "anchor" sets of consecutive columns where a same (short) pattern seems to appear in all sequences. Recursively hinging a global solution around these anchors gives a handle for divide and conquer heuristics. The discovery of anchor patterns fits somewhat into the paradigm of association rule generation. These patterns can be sought among the substrings or subsequences of the sequences, or combinations thereof. For example, one could use the labeling of Karp, Miller, and Rosenberg to label substrings and then look for regions with a concentration of identical labels. A variation on this theme is due to Sagot et al. [122] and is based on the notion of a *model* (direct product of subsets of the alphabet) that extends the notion of a consensus sequence. Models capture the similarity between some categories of symbols as is the case with aminoacids in the comparison of proteins. For fixed lengths, there is a linear-time algorithm to generate all the models common to a set of strings on the basis of hypotheses on two parameters: a *quorum* for the number of implied sequences, and the maximum acceptable number of errors between the models and their actual occurrences.

References

[1] K. Abrahamson. Generalized string matching. *SIAM J. Computing*, 16(6):1039–1051, 1987.

[2] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *Proc. ACM SIGMOD*, pp.207–216, Washington DC May 1993.

[3] A.V. Aho and M.J. Corasick. Efficient string matching. *C. ACM*, 18(6):333–340, 1975.

[4] B. Alberts, D. Bray, J. Lewis, M. Raff, K. Roberts and J.D. Watson. *Molecular Biology of the Cell.* Garland Publishing, N.Y., 1989.

[5] S. Altschul, W. Gish, W. Miller, E.W. Myers and D. Lipman. Basic Linear alignment search tool. *J. Mol. Biology* **215**, 403–410 (1990).

[6] S.F. Altschul and D.J. Lipman. Tree, stars, and multiple biological sequence alignment. *SIAM J. Appl. Math.* 49:197–209, 1989.

[7] A. Amir, A. Apostolico and M. Lewenstein. Inverse Pattern Matching. *J. of Algorithms*, 1997, Vol. 24, No. 2, pp. 325–339.

[8] A. Amir, G. Benson and M. Farach. Let sleeping files lie: pattern matching in Z-compressed files. In *Proc. of 5th Annual ACM-SIAM Symposium on Discrete Algorihms.* 1994.

[9] A. Amir, M. Farach and G. Benson. Let Sleeping Files Lie: Pattern Matching in Z-Compressed Files. *Journal of Computer and System Sciences*, 1996, Vol. 52, No. 2, pp. 299–307.

[10] A. Amir, R. Feldman and R. Kashi. A New and Versatile Method for Association Generation. *Information Systems*, to appear, (preliminary version appeared in PKDD 97).

[11] A. Apostolico. The Myriad Virtues of Subword Trees. In A. Apostolico and Z. Galil, editors, *Combinatorial Algorithms on Words*, volume 12 of *NATO ASI Series F*, pages 85–96. Springer-Verlag, Berlin, Germany, 1985.

[12] A. Apostolico. Optimal Parallel Detection of Squares in Strings. *Algorithmica*, 8:285–319, 1992.

[13] A. Apostolico. String Editing and Longest Common Subsequences. (INVITED PAPER), *Handbook of Formal Languages* (G. Rozenberg and A. Salomaa, Eds.), Vol II, pp. 361–398 Springer-Verlag (1996).

[14] A. Apostolico and G. Bejerano. Optimal Amnesic Probabilistic Automata or How to Learn and Classify Proteins in Linear Time and Space. To appear, *Proceedings of RECOMB 2000*, Tokyo, pp. 25–32 (2000).

[15] A. Apostolico, M.E. Bock, S. Lonardi and X. Xu. Efficient Detection of Unusual Words. Technical Report 97–050, Purdue University Computer Science Department (1996). *Journal of Computational Biology*, in press.

[16] A. Apostolico and D. Breslauer. Of Periods, Quasiperiods, Repetitions and Covers. In *Structures in Logic and Computer Science: A Collection of Essays in Honor of A. Ehrenfeucht*, J. Mycielski, G. Rozenberg and A. Salomaa, Eds., number 1261 in Lecture Notes in Computer Science, pages 236–248. Springer-Verlag, Berlin, Germany, 1992.

[17] A. Apostolico, D. Breslauer, and Z. Galil. Optimal Parallel Algorithms for Periods, Palindromes and Squares. In *Proc. 19th International Colloquium on Automata, Languages, and Programming*, number 623 in Lecture Notes in Computer Science, pages 296–307. Springer-Verlag, Berlin, Germany, 1992.

[18] A. Apostolico and A. Ehrenfeucht. Efficient Detection of Quasiperiodicities in Strings. *Theoret. Comput. Sci.*, 119:247–265, 1993.

[19] A. Apostolico, M. Farach, and C.S. Iliopoulos. Optimal Superprimitivity Testing for Strings. *Inform. Process. Lett.*, 39:17–20, 1991.

[20] A. Apostolico and Z. Galil, Eds. *Pattern Matching Algorithms.* Oxford University Press, New York (1997).

[21] A. Apostolico and R. Giancarlo. Sequence Alignment in Molecular Biology. *Journal of Computational Biology*, **5**, 2:173–196 (1998).

[22] A. Apostolico and F. P. Preparata. Optimal off-line detection of repetitions in a string. *Theoret. Comput. Sci.*, 22:297–315, 1983.

[23] A. Apostolico and F. P. Preparata. Data structures and algorithms for the strings stastitics problem. *Algorithmica*, 15(5):481–494, May 1996.

[24] A. Apostolico and W. Szpankowski. Self-alignment in words and their applications. *J. Algorithms*, 13(3):446–467, 1992.

[25] R. Ash. *Information Theory.* Tracts in mathematics, Interscience Publishers, J. Wiley & Sons, 1985.

[26] R. Baeza-Yates and C. Perleberg. Fast and practical approximate string matching. *Proc. III Symp. on Combinatorial Pattern matching*, Springer LNCS, 185–92 (1992).

[27] M. P. Béal. *Codage Symbolique.* Masson, 1993.

[28] M.-P. Béal, F. Mignosi and A. Restivo. Minimal Forbidden Words and Symbolic Dynamics. In (*STACS'96*, C. Puech and R. Reischuk, eds., LNCS 1046, Springer, 1996) 555–566.

[29] G. Bejerano and G. Yona. Modeling Protein Families Using Probabilistic Suffix Trees. *Proceedings of RECOMB99* (S. Istrail, P. Pevzner and M. Waterman, eds.), 15–24, Lyon, France, ACM Press (April 1999).

[30] T. C. Bell, J. G. Cleary, I. H. Witten. *Text Compression.* Prentice Hall, 1990.

[31] D. R. Bean, A. Ehrenfeucht and G.F. McNulty. Avoidable patterns in strings of symbols. *Pacific J. Math.*, 85:261–294, 1979.

[32] A. Ben-Amram, O. Berkman, C. Iliopolous and K. Park. Computing the Covers of a String in Linear Time. In *Proc. 5th ACM-SIAM Symp. on Discrete Algorithms*, pages 501–510, 1994.

[33] J. Bentley and D. McIlroy. Data compression using long common strings. In *Proceedings of the IEEE Data Compression Conference*, Mar. 1999, pp. 287–295.

[34] J. Berstel. Sur les mots sans carré définis par un morphism. In *Proc. 6th International Colloquium on Automata, Languages, and*

Programming, number 71 in Lecture Notes in Computer Science, pages 16–25. Springer-Verlag, Berlin, Germany, 1979.

[35] J. Berstel. Fibonacci Words — a Survey. In (*The Book of L*, G. Rozenberg, A. Salomaa, eds., Springer Verlag, 1986).

[36] J. Berstel and D. Perrin. Finite and infinite words. In (*Algebraic Combinatorics on Words*, J. Berstel, D. Perrin, eds., Cambridge University Press, to appear) Chapter 1. Available at URL http://www-igm.univ-mlv.fr/~berstel.

[37] A. Blumer, J. Blumer, A. Ehrenfeucht, D. Haussler, M.T. Chen and J. Seiferas. The Smallest Automaton Recognizing the Subwords of a Text. *Theoretical Computer Science*, 40:31–55, 1985.

[38] A. Blumer, A., J. Blumer, A. Ehrenfeucht, D. Haussler, and R. McConnell. Complete Inverted Files for Efficient Text Retrieval and Analysis. *Journal of the ACM*, 34(3): 578–595, 1987.

[39] A. Brazma, J. Vilo, E. Ukkonen and K. Valtonen. Data Mining for Regulatory Elements in Yeast Genome. *Fifth International Conference on Intelligent Systems for Molecular Biology*, ISMB-97 (pp. 65–74) June, 1997. AAAI Press.

[40] A. Brazma, I. Jonassen, I. Eidhammer and D. Gilbert. Approaches to the Automatic Discovery of Patterns in Biosequences. *Journal of Computational Biology* **5**:2, 279–306 (1998).

[41] A. Brazma, I. Jonassen, J. Vilo and E. Ukkonen Pattern Discovery in Biosequences. *Proceedings of Fourth International Colloquium on Grammatical Inference* (ICGI-98) (1433) (pp. 255–270) July 1998. Springer.

[42] A. Brazma, I. Jonassen, J. Vilo and E. Ukkonen. Predicting Gene Regulatory Elements in Silico on a Genomic Scale. *Genome Research* Vol. 8, Issue 11 (pp. 1202–1215) November 1998.

[43] R.P. Brent. Evaluation of General Arithmetic Expressions. *J. Assoc. Comput. Mach.*, 21:201–206, 1974.

[44] D. Breslauer. An On-Line String Superprimitivity Test. *Inform. Process. Lett.*, 44(6):345–347, 1992.

[45] D. Breslauer. Testing String Superprimitivity in Parallel. *Inform. Process. Lett.*, 49(5):235–241, 1994.

[46] D. Breslauer and Z. Galil. A Lower Bound for Parallel String Matching. *SIAM J. Comput.*, 21(5):856–862, 1992.

[47] D. Breslauer and Z. Galil. Finding all Periods and Initial Palindromes of a String in Parallel. *Algorithmica*, 1995.

[48] L. Brillouin. *Science and Information Theory.* Academic Press (1971).

[49] G.S. Brodal, R. Lyngso, C.N.S. Pedersen and J. Stoye. Finding Maximal Pairs with Bounded Gap. *Proc. 10th Combinatorial Pattern Matching*, 342–351. Springer Verlag LNCS volume 1645 (1999).

[50] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Tech. Rep. 124, Digital Equipments Corporation, May 1994.

[51] H. Carillo and D.J. Lipman. The multiple sequence alignment problem in biology. *SIAM J. Appl. Math.* 48:1073–1083, 1988.

[52] S.C. Chan, A.K. Wong and D.K. Chiu. A survey of multiple sequence comparison methods. *Bull. Math. Biol.* 54:563–598, 1992.

[53] W.I. Chang and E.L. Lawler. Sublinear expected time approximate string matching and biological applications. *Algorithmica* **12**, 327–44 (1994).

[54] M.T. Chen and J. Seiferas. Efficient and Elegant Subword-tree Construction. In A. Apostolico and Z. Galil, editors, *Combinatorial Algorithms on Words*, volume 12 of *NATO ASI Series F*, pages 97–107. Springer-Verlag, Berlin, Germany, 1985.

[55] C. Choffrut and K. Culik. On Extendibility of Unavoidable Sets. *Discrete Appl. Math.* **9**, 1984, 125–137.

[56] R. Cole, M. Crochemore, Z. Galil, L. Gąsieniec, R. Hariharan, S. Muthukrishnan, K. Park and W. Rytter. Optimally Fast Parallel Algorithms for Preprocessing and Pattern Matching in One and Two Dimensions. In *Proc. 34th IEEE Symp. on Foundations of Computer Science*, pages 248–258, 1993.

[57] M. Crochemore. An optimal algorithm for computing the repetitions in a word. *Inform. Process. Lett.*, 12(5):244–250, 1981.

[58] M. Crochemore. Sharp characterizations of squarefree morphisms. *Inform. Process. Lett.*, 18:221–226, 1982.

[59] M. Crochemore. Optimal Factor Transducers. In A. Apostolico and Z. Galil, editors, *Combinatorial Algorithms on Words*, volume 12 of *NATO ASI Series F*, pages 31–43. Springer-Verlag, Berlin, Germany, 1985.

[60] M. Crochemore. Transducers and repetitions. *Theoret. Comput. Sci.*, 12:63–86, 1986.

[61] M. Crochemore and C. Hancart. Automata for matching patterns. In (*Handbook of Formal Languages*, G. Rozenberg, A. Salomaa, eds., Springer-Verlag, 1997, Volume 2, *Linear Modeling: Background and Application*) Chapter 9, 399–462.

[62] M. Crochemore, F. Mignosi and A. Restivo. Minimal Forbidden Words and Factor Automata. In (*MFCS'98*, L. Brim, J. Gruska, J. Slatuška, eds., LNCS 1450, Springer, 1998) 665–673.

[63] M. Crochemore, F. Mignosi and A. Restivo. Automata and Forbidden Words. *Information Processing Letters* 67 (1998) 111–117.

[64] M. Crochemore, F. Mignosi, A. Restivo and S. Salemi. Text Compression Using Antidictionaries. Tech. Rept. IGM-98–10, Institut Gaspard Monge, 1998. DCA home page at URL http://www-igm.univ-mlv.fr/~mac/DCA.html

[65] M. Crochemore and W. Rytter. *Text Algorithms*, Oxford University Press, New York (1994).

[66] M. Crochemore and W. Rytter. Efficient parallel algorithms to test square-freeness and factorize strings. *Inform. Process. Lett.*, 38:57–60, 1991.

[67] M. Crochemore and W. Rytter. Usefulness of the Karp-Miller-Rosenberg algorithm in parallel computations on strings and arrays. *Theoret. Comput. Sci.*, 88:59–82, 1991.

[68] G. Das, R. Fleischer, L. Gąsieniek, D. Gunopulos and J. Kärkkäinen. Episode Matching. *CPM'97, Proceedings of the 8th Annual Symposium on Combinatorial Pattern Matching*, (A. Apostolico and J. Hein, Eds.), Springer Verlag LNCS **1264**, 12–27 (1997).

[69] V. Diekert and Y. Kobayashi. *Some Identities Related to Automata, Determinants, and Möbius Functions.* Report Nr. 1997/05, Universität Stuttgart, Fakultät Informatik, 1997.

[70] R. S. Ellis. *Entropy, Large Deviations, and Statistical Mechanics.* Springer Verlag, 1985.

[71] M. Farach, M. Thorup. String matching in Lempel-Ziv compressed strings. In *Proc. of 27th Symposium on Theory of Computing* 1994, 703–713.

[72] P. Ferragina. Dynamic Data Structures for String Matching Problems. *Doctoral Thesis*, University of Pisa (1997).

[73] E. R. Fiala and D. H. Greene. Data compression with finite windows. *Communications of the ACM*, vol. 32, pp. 490–505, 1989.

[74] F.E. Fich, R.L. Ragde and A. Wigderson. Relations Between Concurrent-write Models of Parallel Computation. *SIAM J. Comput.*, 17(3):606–627, 1988.

[75] N.J. Fine and H.S. Wilf. Uniqueness Theorems for Periodic Functions. *Proc. Amer. Math. Soc.*, 16:109–114, 1965.

[76] M.J. Fischer and M.S. Paterson. String matching and other products. *Complexity of Computation, R.M. Karp (editor), SIAM-AMS Proceedings*, 7:113–125, 1974.

[77] K. S. Fu and T. L. Booth. Grammatical inference: Introduction and survey – Part I. *IEEE Transactions on Systems, Man and Cybernetics*, 5:95–111, 1975.

[78] K. S. Fu and T. L. Booth. Grammatical inference: Introduction and survey — Part II. *IEEE Transactions on Systems, Man and Cybernetics*, 5:112–127, 1975.

[79] J. Gailly. *Frequently Asked Questions in data compression.* At URL http://www.landfield.com/faqs/compression-faq/

[80] L. Gąsieniec, P. Indyk and P. Krysta. External Inverse Pattern Matching. *Proceedings of the 8th Annual Symposium on Combinatgorial Pattern Matching*, Springer-Verlag LNCS 1264, pp. 90–101 (1997).

[81] L. Gąsieniec and W. Rytter. Almost optimal fully LZW-compressed pattern matching. In *Data Compression Conference*, J. Storer, ed, 1999.

[82] L. Gatlin. *Information Theory and the Living Systems.* Columbia University Press, 1972.

[83] D. Greene, M. Parnas and F. Yao. Multi-index hashing for information retrieval. *Proc. 35th Annual Symposium on Foundations of Computer Science*, pages 722–731, 1994.

[84] D. Gusfield and J. Stoye. Simple and Flexible Detection of Contiguous Repeats Using a Suffix Tree. *9th CPM 98*, Springer LNCS 1448, 140–152.

[85] M. Gu, M. Farach and R. Beigel. An efficient algorithm for dynamic text indexing. In *Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, Arlington, VA, 1994, pp. 697–704.

[86] D. Gusfield and J. Stoye. Linear Time Algorithms for Finding and Representing all Tandem Repeats in a String. Technical report CSE-98–4, UC Davis Computer Science. 1998.

[87] R. W. Hamming. Error detecting and error correcting codes. *Bell System Tech. J.*, 29:147–160, 1950.

[88] R. N. Horspool. The effect of non-greedy parsing in Ziv-Lempel compression methods. In *DCC: Data Compression Conference.* 1995, pp. 302–311, IEEE Computer Society TCC.

[89] C.S. Iliopoulos, D.W.G. Moore and K. Park. Covering a String. In *Proc. 4th Symp. on Combinatorial Pattern Matching*, number 684 in Lecture Notes in Computer Science, pages 54–62, Berlin, Germany, 1993. Springer-Verlag.

[90] C.S. Iliopoulos and K. Park. An Optimal $O(\log \log n)$-time Algorithm for Parallel Superprimitivity Testing. *J. Korea Information Science Society*, 21(8):1400–1404, 1994.

[91] R. Karp and M.O. Rabin. Efficient Randomized Pattern Matching Algorithms. *IBM J. Res. Dev.* **31**, 249–260 (1987).

[92] J. G. Kemeny and J. L. Snell. *Finite Markov Chains.* Van Nostrand Reinhold, 1960.

[93] D.E. Knuth, J.H. Morris and V.R. Pratt. Fast Pattern Matching in Strings. *SIAM J. Comput.*, 6:322–350, 1977.

[94] A. N. Kolmogorov. Three approaches to the quantitative definition of information. *Problemi Pederachi Inf.*, 1, 1965.

[95] S. Kurtz. Reducing the space requirments of suffix trees. Tech. Rep. 98–03, Technischen Fakultät, Universität Bielefeld, 1998.

[96] N. J. Larsson. Extended application of suffix trees to data compression. In *DCC: Data Compression Conference*. 1996, pp. 190–199, IEEE Computer Society TCC.

[97] A. Lempel and J. Ziv. On the complexity of finite sequences. *IEEE Trans. on information Theory*, 22:75–81, 1976.

[98] V.I. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Phys. Dokl.*, 6:707–710, 1966.

[99] M. Lothaire. *Combinatorics on Words.* Cambridge University Press, second edition, 1997.

[100] R. C. Lyndon and M. P. Schutzenberger. The equation $a^m = b^n c^p$ in a free group. *Michigan Math. J.*, 9:289–298, 1962.

[101] M.G. Main and R.J. Lorentz. An $O(n \log n)$ algorithm for finding all repetitions in a string. *J. of Algorithms*, pages 422–432, 1984.

[102] G. Manacher. A new Linear-Time On-Line" Algorithm for Finding the Smallest Initial Palindrome of a String. *J. Assoc. Comput. Mach.*, 22, 346–351, 1975.

[103] U. Manber. A text compression scheme that allows fast searching directly in the compressed file. In M. Crochemore and D. Gusfield, editors, *Proceedings of the 5th Annual Symposium on Combinatorial Pattern Matching*, number 807 in Lecture Notes in Computer Science, pages 113–124, Asilomar, CA, 1994. Springer-Verlag, Berlin.

[104] U. Manber and R. Baeza-Yates. An algorithm for string matching with a sequence of don't cares. *Inform. Process. Lett.* 37 (1991), No. 3, 133–136.

[105] U. Manber and E. Myers. Suffix Arrays: a New Method for On-line String Searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

[106] H. Mannila, H. Toivonen and A.I. Vercamo. Discovering Frequent Episodes in Sequences. *KDD'95, Proceedings of the 1st International Conference on Knowledge Discovery and Data Mining*, AAAI Press, 210–215 (1995).

[107] P. Martin-Lof. The definition of random sequences. *Information and Control*, 9, 1966.

[108] E.M. McCreight. A Space Economical Suffix Tree Construction Algorithm. *J. Assoc. Comput. Mach.*, 23:262–272, 1976.

[109] D. Moore and W.F. Smyth. Computing the Covers of a String in Linear Time. In *Proc. 5th ACM-SIAM Symp. on Discrete Algorithms*, pages 511–515, 1994.

[110] M. Morse and G. Hedlund. Symbolic Dynamics II: Sturmian trajectoires. *Amer. J. Math.* 62 (1940) 1–40.

[111] E. Moura, G. Navarro, N. Ziviani and R. Beaza-Yates. Direct pattern matching on compressed texts. In *Proc. SPIRE'98*, IEEE CS Press, 1998, 90–95.

[112] S. Muthukrishnan. Non-standard Stringology: Algorithms and Complexity. *Proc. 26th Annual Symposium on the Theory of Computing*, pages 770–779, 1994.

[113] G. Navarro and M. Raffinot. Pattern matching in compressed texts. To appear.

[114] M. Nelson and J. Gailly. *The Data Compression Book.* M&T Books, New York, NY, 1996. 2nd edition.

[115] G. Piatesky-Shapiro and W.J. Frawley, Eds. *Knowledge Discovery in Databases.* AAAI Press/MIT Press, 1991.

[116] K.R. Popper. *The Logic of Scientific Discovery.* Hutchinson, London, 1959.

[117] M. Rabin. Discovering Repetitions in Strings. In *Combinatorial Algorithms on Words* (A. Apostolico and Z. Galil, eds.), Springer Verlag pp. 279–288 (1985).

[118] J. Rissanen. A universal Data Compression System. *IEEE Trans. Inform. Theory* **29**(5): 656–664 (1983).

[119] J. Rissanen. Complexity of Strings in the Class of Markov Sources. *IEEE Trans. Inform. Theory* **32**(4): 526–532 (1986).

[120] D. Ron, Y. Singer and N. Tishby. The Power of Amnesia: Learning Probabilistic Automata with Variable Memory Length. *Machine Learning*, 25:117–150 (1996).

[121] D. Russel and G.T. Gangemi, Sr. *Computer Security Basics.* O'Reilly and Associates, Inc., Sebastopol, California, 1991.

[122] M.-F. Sagot, A. Viari and H. Soldano. Multiple sequence comparison — A peptide matching approach. *Theoret. Comput. Sci.* 180(1–2):115–137, 1997.

[123] C.E. Shannon and W. Weaver. *The Mathematical Theory of Communication.* University of Illinois Press, Urbana (1949).

[124] C.E. Shannon. Prediction and entropy of printed english. *Bell System Technical J.*, 50–64, January, 1951.

[125] Y. Shibata, M. Takeda, A. Shinohara and S. Arikawa. Pattern matching in text compressed by using antidictionaries. In *Combinatorial Pattern Matching*, M. Crochemore and M. Paterson, eds, number 1645 in LNCS, Springer, 1999, 37–49.

[126] J.A. Storer. *Data Compression: Methods and Theory.* Computer Science Press, 1988.

[127] J.A. Storer and T. G. Szymanski. Data compression via textual substitution. *Journal of the ACM*, 29(4):928–951, October 1982.

[128] A. Thue. Über unendliche zeichenreihen. *Norske Vid. Selsk. Skr. Mat. Nat. Kl. (Cristiania)*, (7):1–22, 1906.

[129] A. Thue. Über die gegenseitige lage gleicher teile gewisser zeichenreihen. *Norske Vid. Selsk. Skr. Mat. Nat. Kl. (Cristiania)*, (1):1–67, 1912.

[130] E. Ukkonen. On-line Construction of Suffix Trees. *Algorithmica*, 14:249–260, 1995.

[131] E. Ukkonen. Approximate string matching and the q-gram distance. In: R. Capocelli, A. De Santis and U. Vaccaro (eds.), *SEQUENCES II - Methods in Communication, Security, and Computer Science*, 300–312, Springer 1993.

[132] R. von Mises. *Probability, Statistics and Truth.* MacMillan, New York, 1939.

[133] S. Watanabe. *Knowing and Guessing.* Wiley, New York, 1969.

[134] M. Waterman. *Introduction to Computational Biology.* Chapman and Hall (1995).

[135] P. Weiner. Linear Pattern Matching Algorithms. In *Proc. 14th Symposium on Switching and Automata Theory*, pages 1–11, 1973.

[136] T.A. Welch. A technique for high performance data compression. *IEEE Trans. on Computers*, 17:8–19, 1984.

[137] I.H. Witten, A. Moffat and T. C. Bell. *Managing Gigabytes.* Van Nostrand Reinhold, 1994.

[138] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Trans. on Inform. Theory*, vol. IT-23, no. 3, 337–343 (1977).

[139] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Trans. on Inform. Theory*, vol. 24, no. 5, 530–536, Sept. 1978.